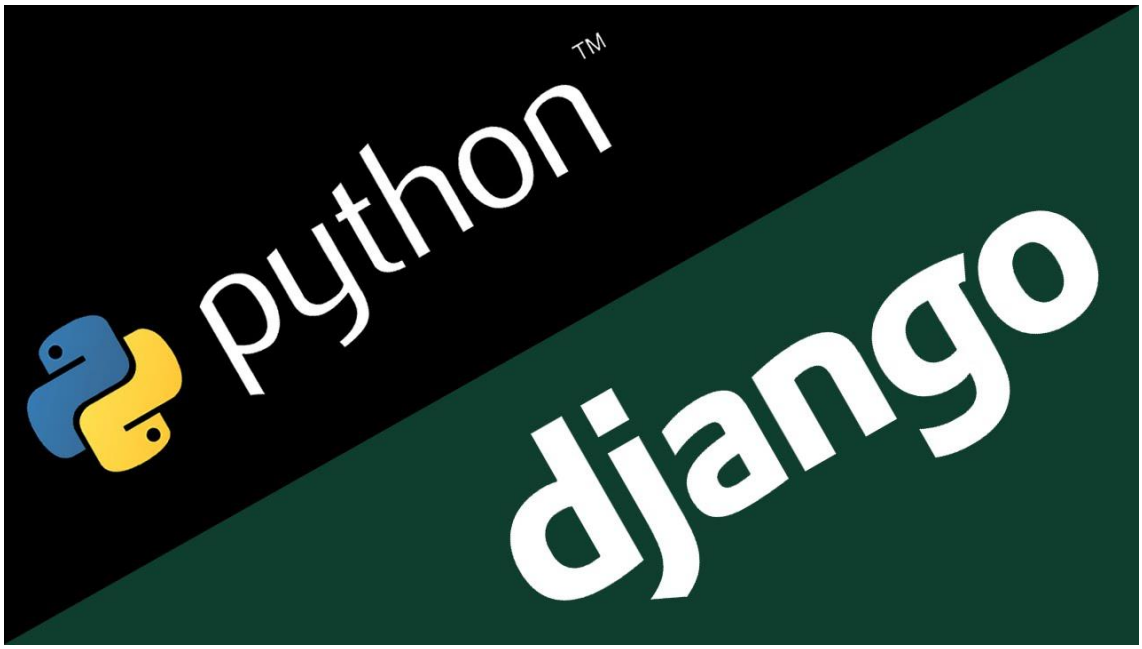
Tutorial Django Framework

Web Developer



Sumário

Apostila de Django Python

Autor: Anderson Henrique R Maciel

Capítulo 1

O Django Framework

Django é um framework de alto nível, escrito em Python que encoraja o desenvolvimento limpo de aplicações web.

Desenvolvido por experientes desenvolvedores, Django toma conta da parte pesada do desenvolvimento web, como tratamento de requisições, mapeamento objeto-relacional, preparação de respostas HTTP, para que, dessa forma, você gaste seu esforço com aquilo que realmente interessa: suas regras de negócio!

Foi desenvolvido com uma preocupação extra em segurança, evitando os mais comuns ataques, como Cross site scripting (XSS), Cross Site Request Forgery (CSRF), SQL injection, entre outros.

É bastante escalável: Django foi desenvolvido para tirar vantagem da maior quantidade de hardware possível. Django usa uma arquitetura "zero compartilhamento", o que significa que você pode adicionar mais recursos em qualquer nível:servidores de banco de dados, cache e/ou servidores de aplicação.

Para termos uma boa noção do Django como um todo, esse curso utiliza uma abordagem bottom-up (de baixo para cima): primeiro veremos os conceitos do Django, depois abordaremos a Camada de Modelos, depois veremos a Camada de Views e, por fim, a Camada de Templates.

Como funciona o Django?

Como disse anteriormente, o Django é um framework para construção de aplicações web em Python. E, como todo framework web, ele é um framework MVC (Model View Controller), certo??? Bem... Não exatamente!

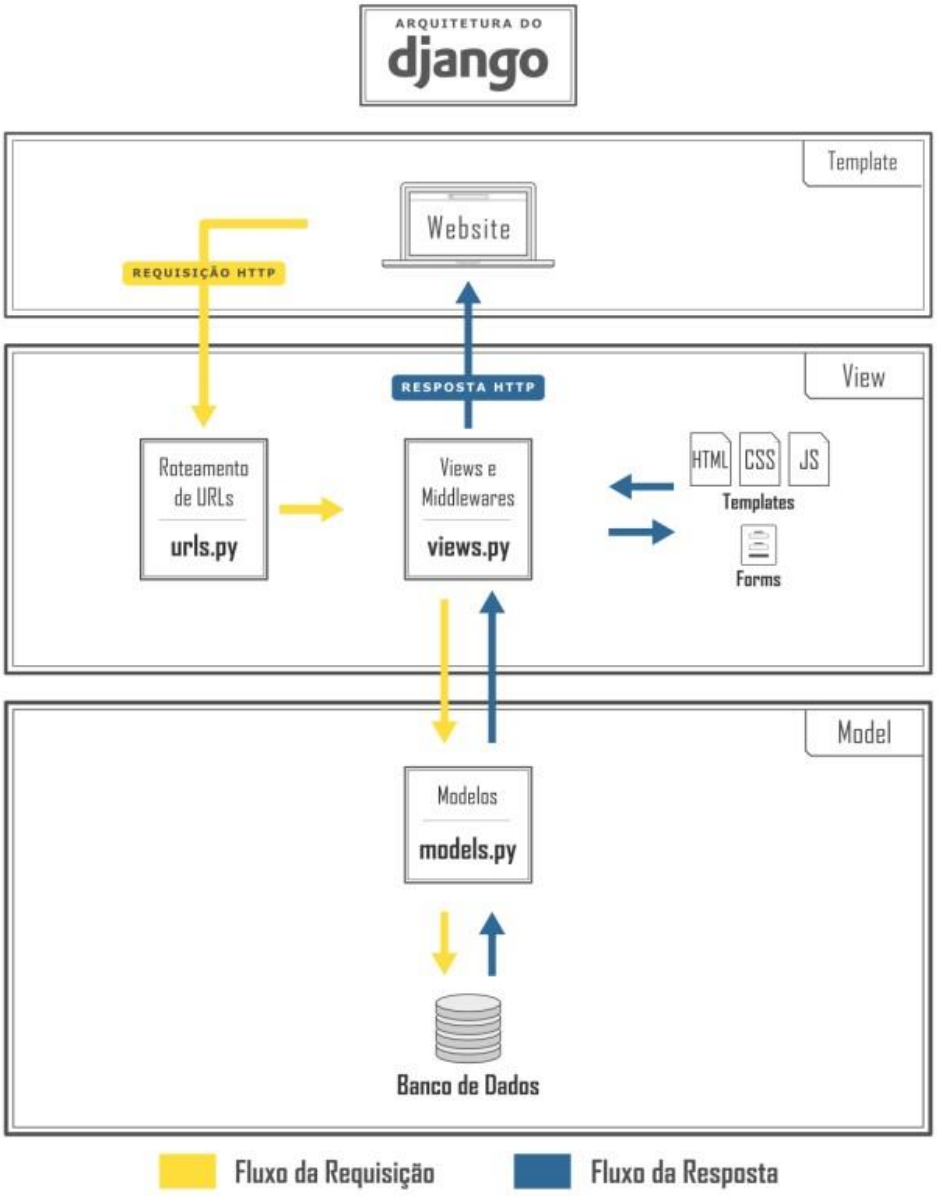
De acordo com sua documentação, os desenvolvedores o declara como um framework MTV - isto é: Model-Template-View. Mas, por que a diferença??? Para os desenvolvedores, as Views do Django representam qual informação você vê, não como você vê. Há uma sutil diferença.

No Django, uma View é uma forma de processar os dados de uma URL específica, pois ela descreve qual informação é apresentada, através do processamento descrito pelo desenvolvedor em seu código. Além disso, é imprescindível separar conteúdo de apresentação - que é onde os templates residem.

Como disse, uma View descreve qual informação é apresentada, mas uma View normalmente delega para um template, que descreve como a informação é apresentada. Assim, onde o Controller se encaixa nessa arquitetura??? No caso do Django, é o próprio framework que faz o trabalho pesado de processar e rotear uma requisição para a View apropriada de acordo com a configuração de URL descrita pelo desenvolvedor.

Fluxo de uma requisição no Django

Para ajudar a entender um pouco melhor, vamos analisar o fluxo de uma requisição saindo do browser do usuário, passando para o servidor onde o Django está sendo executado e retornando ao browser do usuário. Veja a seguinte ilustração:



O Django é dividido em três camadas:

- A Camada de Modelos
- A Camada de Views
- A Camada de Templates

Vamos agora, dar nossos primeiros passos com o Django começando pela sua instalação!

Introdução (Criando o Ambiente Virtual)

A boa prática de desenvolvimento com python é a utilização de um ambiente virtual, consiste em isolar a instalação do python para o projeto que será desenvolvido, não utilizaremos os pacotes instalados globalmente, presentes no Sistema Operacional.

- Instalar o Python: Instalador Python do site
- Abrir o terminal e acessar a pasta que deseja criar o ambiente virtual
- Após acessar a pasta digite o comando: python -m venv nome_ambiente (geralmente o nome é venv)
- Após criar o ambiente virtual, precisamos ativar ele, acesse a pasta do ambiente virtual pelo CMD venv\Scripts\Activate.bat
- Se o ambiente virtual foi executado mostrará no início da linha de comando o nome do ambiente (venv) C:\
- Se quiser desativar o ambiente virtual digite: deactivate

Instalar o Django no ambiente virtual

- Com o ambiente ativado, no diretório do ambiente virtual digite o comando: pip install django==5.0 (versão LTS mais estável)
- Para saber se o Django está instalado no ambiente virtual, digite o comando: python
- No python digite o comando: import django
- No python digite o comando: print(django.get_version()) e exit()

Criando o projeto no ambiente virtual com Django

- Com o ambiente virtual ativado, acesse o diretório do ambiente para criar o projeto e digite: django-admin startproject nome_projeto (helloworld)
- Verifique dentro da pasta do ambiente, se criou o diretório do projeto

Configurando o Banco de Dados

- Abrir o arquivo settings.py e verificar em DATABASES que o nosso banco de dados padrão é o sqlite3
- Precisamos fazer as migrações necessárias para o projeto: python manage.py migrate

Testando o servidor local

- Acesse o diretório do projeto e digite: python manage.py runserver
- A porta utilizada para o servidor é: 127.0.0.1:8000, parar CTRL + Break

Criando superuser (usuário admin)

- Para criar um usuário de administrador digite: python manage.py createsuperuser. Defina um Username, Email, Password e Confirm Password

Criando a nossa primeira aplicação

- O ideal é que a nossa aplicação seja criada no nosso diretório do projeto raiz, digite: python manage.py startapp nome_aplicação (website)

Vamos organizar a estrutura da nossa aplicação

- Dentro da pasta da aplicação (website), crie a pasta templates. Dentro dela crie uma subpasta website, e dentro da subpasta uma pasta _layouts
- Crie também a pasta static dentro da pasta da aplicação (website), para guardar os arquivos estáticos (CSS, Javascript, imagens, fontes, etc)
- Dentro dela crie uma pasta website, por questões de namespace. Dentro dela crie: uma pasta css, uma pasta img e uma pasta js

Adicionando app a lista de apps instalados

- Para que o django gerencie esse app, é necessário adicioná-lo à lista de apps instalados.
- Fazemos isso atualizando a configuração INSTALLED_APPS no arquivo de configuração helloworld/settings.py
- Adicione website e helloworld

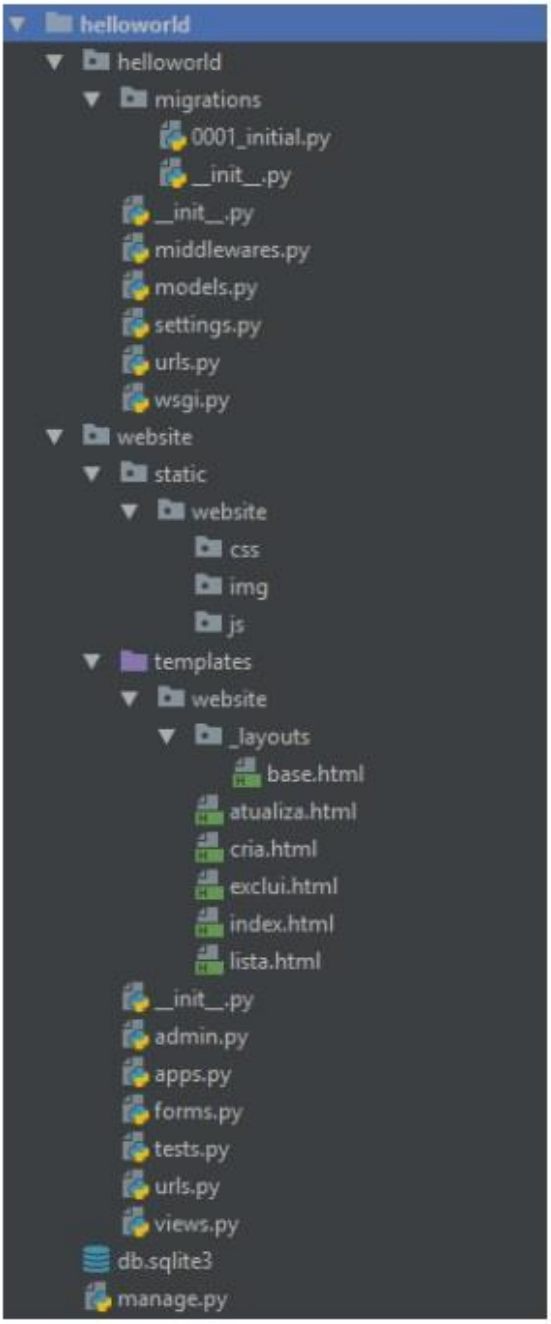
  'helloworld',

  'website'

Alterações na estrutura do projeto

- Primeiro, vamos passar o arquivo de modelos models.py de /website para /helloword, pois os arquivos comuns ao projeto vão ficar centralizados no app helloworld (geralmente temos apenas um arquivo models.py para o projeto todo)

- Como não temos mais o arquivo de modelos na pasta /website, podemos excluir a pasta /migrations e o migrations.py, pois estes serão gerados e gerenciados pelo app helloworld



A Camada Model

A Camada de Modelos tem uma função essencial na arquitetura das aplicações desenvolvidas com o Django. É nela que descrevemos os campos e comportamentos das

entidades que irão compor nosso sistema. Também é nela que reside a lógica de acesso aos dados da nossa aplicação.

Vamos mergulhar um pouco mais e conhecer a camada Model da arquitetura MTV (Model-Template-View) do Django. Nela, vamos descrever, em forma de classes, as entidades do nosso sistema, para que o resto (Template e View) façam sentido.

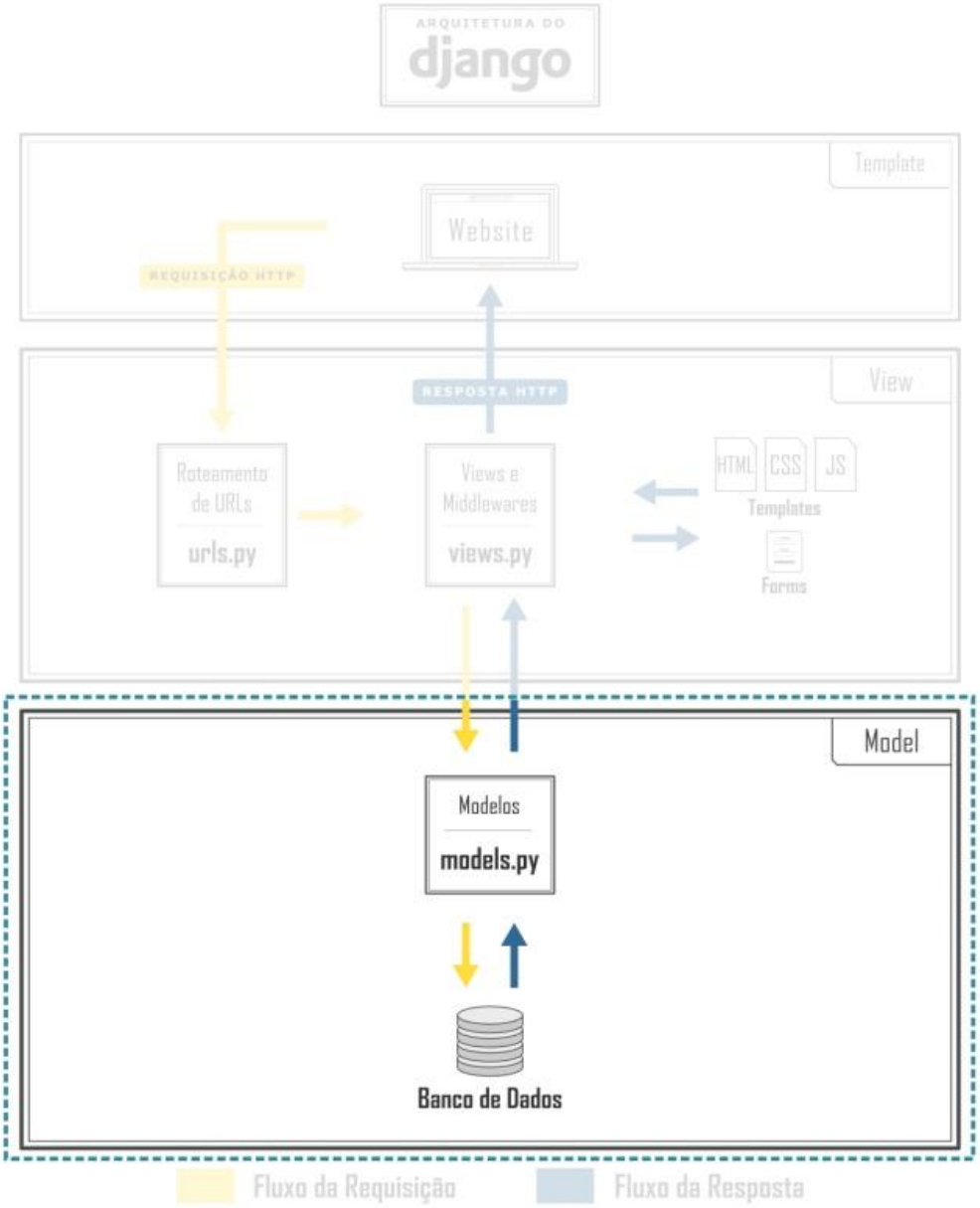
Vamos começar pelo básico: pela definição de modelo!

Um modelo é a descrição do dado que será gerenciado pela sua aplicação. Ele contém os campos e comportamentos desses dados. No fim, cada modelo vai equivaler à uma tabela no banco de dados.

No Django, um modelo tem basicamente duas características:

- É uma classe que herda de django.db.models.Model
- Cada atributo representa um campo da tabela

Com isso, Django gera automaticamente uma API de Acesso à Dados. Essa API facilita e muito nossa vida quando formos gerenciar (adicionar, excluir e atualizar) nossos dados.



Exemplo: Vamos modelar nosso "Hello World!"

Vamos supor que sua empresa está desenvolvendo um sistema de gerenciamento dos funcionários e lhe foi dada a tarefa de modelar e desenvolver o acesso aos dados da entidade Funcionário

Pensando calmamente em sua estação de trabalho enquanto seu chefe lhe cobra diversas metas e dizendo que o deadline do projeto foi adiantado em duas semanas você pensa nos seguintes atributos para tal classe:

- Nome
- Sobrenome
- CPF
- Tempo de Serviço
- Remuneração

OK!

Agora é necessário passar isso para o código Python para que o Django possa entender. No Django os modelos são descritos no arquivo models.py. Esse arquivo já foi criado e está presente na pasta helloworld/models.py

Nele, nós iremos descrever cada atributo (nome, sobrenome, CPF e etc) como um campo (ou Field) da nossa classe de Modelo, vamos chamar essa classe de Funcionario, seguindo as duas características:

1. Herdar da classe Model (from django.db import models)
2. Mapear os atributos da entidade com os campos (ORM)

```
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
    )
    objetos = models.Manager()
```

Agora que criamos nosso modelo, é necessário executar a criação das tabelas no banco de dados. Para isso, o Django possui dois comandos que ajudam muito: o makemigrations e o migrate.

O comando makemigrations analisa se foram feitas mudanças nos modelos e, em caso positivo, cria novas migrações (Migrations) para alterar a estrutura do seu banco de dados, refletindo as alterações feitas. Portanto, toda vez que você alterar o seu modelo, não se esqueça de executar: manage.py makemigrations (aplicação)

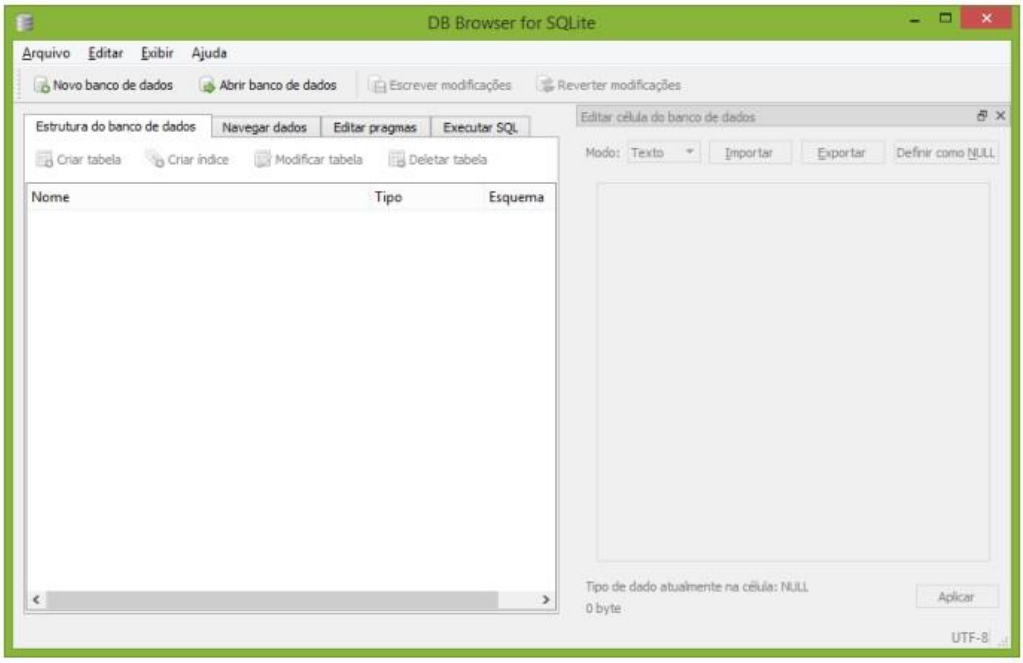
O comando migrate quando executamos, o Django cria o banco de dados e as migrations, mas não as executa, isto é: não aplica as alterações no banco de dados. Se você criou o projeto com django-admin.py createproject helloworld, a configuração padrão foi aplicada. Procure pela configuração DATABASES no settings.py. Por padrão, o Django utiliza um banco de dados leve e completo chamado SQLite

Agora vamos executar o comando migrate, propriamente dito! python manage.py migrate

CALMA lá... Havíamos definido apenas uma Migration e foram aplicadas 15!!! Por quê??? Se lembra que a configuração INSTALLED_APPS continha vários apps (e não apenas o nosso helloworld e website)? Pois então, cada app desses contém seus próprios modelos e migrations. ENTENDEU??? Isto está muito abstrato! Como eu posso ver o banco, as tabelas e os dados na prática?

Ferramenta DB Browser for SQLite

Apresento uma ferramenta MUITO prática que nos auxilia verificar se nosso código está fazendo aquilo que queríamos. Com ela, podemos ver a estrutura do banco de dados, alterar dados em tempo real, fazer queries, verificar se os dados foram efetivados no banco e muito mais! Site: https://sqlitebrowser.org/



Após a instalação, execute o programa e clique em Abrir banco de dados e procure pelo banco de dados no nosso projeto db.sqlite3, ao importá-lo teremos uma visão geral, mostrando Tabelas, Índices, Views e Triggers.

Para ver os dados de cada tabela, vá para a aba "Navegar Dados", escolha a nossa tabela helloworld_funcionario e... ESTÁ VAZIA, ainda não adicionamos nada! Já já vamos criar as Views e Templates e popular esse DB!

API de Acesso a Dados (ORM – Object-Relational-Mapping)

Com nossa classe Funcionario modelada, vamos agora ver a API de acesso à dados provida pelo Django para facilitar e muito a nossa vida! Vamos testar a adição de um novo funcionário utilizando o shell do Django. Digite o comando: python manage.py shell

O shell do Django é muito útil para testar trechos de código sem ter que executar o servidor inteiro! Para adicionar um novo funcionário, basta criar uma instância do seu modelo e chamar o método save(), um método herdado da classe Models, digite:

```
from helloworld.models import Funcionario
funcionario = Funcionario(
        nome='Marcos',
        sobrenome='da Silva'
        cpf='015.458.895-50',
        tempo_de_servico=5,
        remuneracao=10500.00
)
```

```
funcionario.save()
funcionario = Funcionario(
        nome='Aline',
        sobrenome='de Sousa'
        cpf='041.485.721-60',
        tempo_de_servico=4,
        remuneracao=7500.00
)
funcionario.save()
```

E pronto! O Funcionário Marcos da Silva será salvo no seu banco! NADA de código SQL e queries enormes!! Tudo simples! Tudo limpo! Tudo PYTHON!, mas COMO ASSIM?

Por exemplo, para buscar todos os funcionários, abra o shell do Django e digite: funcionarios = Funcionario.objetos.all()

O Manager que utilizamos na criação do Model Funcionario é a interface na qual as operações de busca são definidas para o seu modelo, ou seja, através do campo objetos podemos fazer queries incríveis sem uma linha de SQL!

Exemplo: Busque todos os funcionários que tenham mais de 3 anos de serviço, que ganhem menos de R$ 5.000,00 de remuneração e que não tenham Marcos no nome:

```
funcionarios = Funcionario.objetos
        .exclude(nome="Marcos")
        .filter(tempo_de_servico__gt=3)
        .filter(remuneracao__lt=5000.00)
        .all()
```

O método exclude() retira linhas da pesquisa e filter() filtra a busca. No filtro, concatenamos a string __gt (gt = greater than = maiores que) e __lt(lt = less than = menores que). O método .all() ao final da query serve para retornar todas as linhas do banco que cumpram os filtros da nossa busca. (Também temos o método first() que retorna o primeiro registro, o last(), que retorna o último).

Agora, vamos ver como é simples excluir um Funcionário:

```
#Primeiro, encontramos o funcionário que desejamos deletar
funcionario = Funcionario
        .objetos
        .filter(id=1)
        .first()
#Agora, o deletamos!
funcionario.delete()
```

Interessante e Fácil!!!

A atualização também é extremamente simples, bastando buscar a instância desejada, alterar o campo e salvá-lo novamente! Por exemplo: o funcionário de id=2 se casou e alterou seu nome de Aline de Sousa para Aline de Sousa Pinheiro, podemos alterar assim:

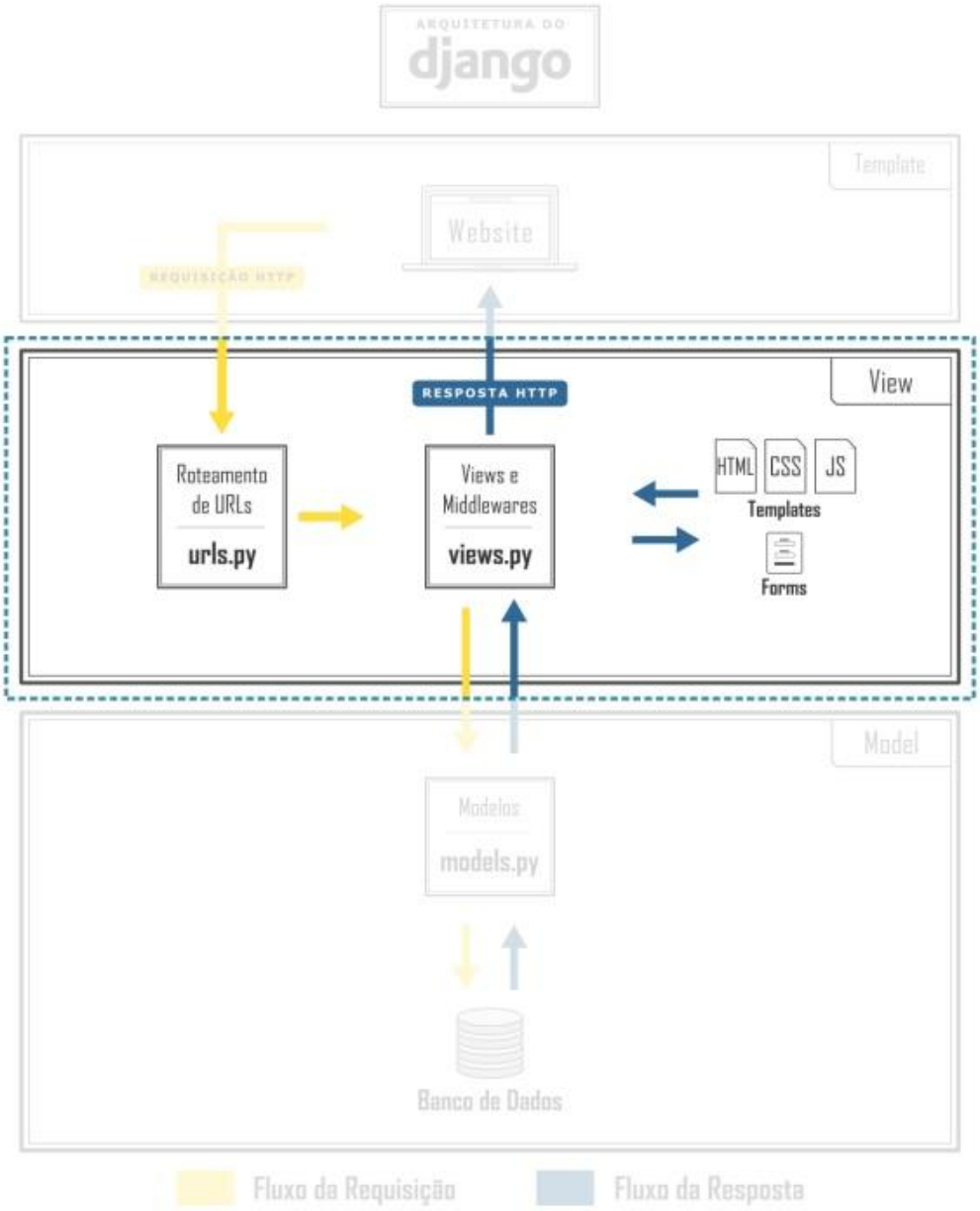
```
#Primeiro buscamos o funcionário desejado
funcionario = Funcionario
        .objetos
        .filter(id=2)
        .first()
#Alteramos o campo sobrenome
funcionario.sobrenome = funcionario.sobrenome + " Pinheiro"
#Salvamos as alterações
funcionario.save()
```

Capítulo 2

A Camada View

Sobre a camada view, é nela que descreveremos a lógica de negócios da nossa aplicação, ou seja: é nela que vamos descrever os métodos que irão processar as requisições, formular respostas e enviá-las de volta ao usuário.

Vamos aprender o conceito das Views do Django, aprender a diferença entre Function Based Views e Class Based Views, como utilizar Forms, aprender o que é um Middleware e como desenvolver nossos próprios e muito mais.



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: recepcionar, 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 a 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 a requisição. Dessa forma, o arquivo helloworld/urls.py deve conter:

```
from django.contrib import admin

from django.urls import path

from django.urls.conf import include


urlpatterns = [

    # Inclui as URLs do app 'website'

    path('', include('website.urls', namespace='website')),

    # Interface administrativa

    path('admin/', admin.site.urls),

]
```

Assim, o Django irá tentar fazer o match (casamento) de URLs primeiro no arquivo de URLs do app WebSite (website/urls.py) depois no URLConf da plataforma administrativa. 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 raiz /, 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 contém a lista de roteamento de URLs

urlpatterns = [

    #GET

    path('', views.index, name='index'),

]
```

O atributo app_name = 'website' define o namespace do app website, o método path() tem a seguinte assinatura: path(rota, view, kwargs=None, name=Nome)

rota: string contendo a rota (URL)

view: a função (ou classe) que irá tratar essa rota

kwargs: utilizado para passar dados adicionais à função ou método que irá tratar a requisição

name: nome da rota. O Django utiliza o app_name mais o nome da rota para nomear a URL. Por exemplo, no nosso caso, podemos chamar a rota raiz '/' com 'website:index' (app_site = website e a rota raiz = index)

No arquivo views.py, vamos criar a função que irá tratar a rota:

```
from django.shortcuts import render

from django.http import HttpResponse


#criando a função index

def index(request) :

    num1 = 2.0

    num2 = 1.0

    soma = num1 + num2

    return HttpResponse('Curso de Django ' + str(soma))
```

Function 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.index(). Podemos tratar as requisições de duas formas: através de funções (Function Based Views) ou através de Class Based Views (ou apenas CBVs). Utilizando funções, você basicamente vai definir uma função que:

1. Recebe como parâmetro uma requisição (request)
2. Realiza algum processamento
3. Retorna alguma informação

Já as Class Based Views são classes que herdam da classe do Django django.view.generic.base.View e que agrupam diversas funcionalidades a vida do desenvolvedor. Nós podemos herdar e estender as funcionalidades das CBVs para atender a lógica da nossa aplicação. Por exemplo, suponha que você quer criar uma página com a listagem de todos os funcionários, utilizando funções, você pode atingir o objetivo da seguinte forma:

```
from helloworld.models import Funcionario

def lista_funcionarios(request) :
    # primeiro, buscamos os funcionários
    funcionarios = Funcionario.objetos.all()

    # incluímos no contexto
    contexto = {
        'funcionarios': funcionarios
    }
    # retornamos o template para listar os funcionários
    return render(
        request,"website/funcionarios.html",contexto
    )
```

Algumas observações:

Toda função que vai processar requisições no Django recebe como parâmetro um objeto request contendo os dados da requisição

Contexto é o conjunto de dados que estarão disponíveis para construção de cada página - ou template

A função Django.shortcuts.render() é um atalho do próprio Django que facilita a renderização de templates: ela recebe a própria requisição, do diretório do template, o contexto da requisição e retorna o template renderizado

No exemplo abaixo, utilizamos o modificador **private** no atributo e método da classe, portanto, podemos acessá-los a partir:

Utilizando CBVs, podemos utilizar a listView presente em django.views.generic para listar todos os funcionários, da seguinte forma:

```
from django.views.generic import ListView

class ListaFuncionarios(ListView) :

    template_name = "website/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 descreveram 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 isso que a ListView faz.

Com isso, um objeto funcionarios estará disponível no seu template para iteração, dessa forma, podemos criar uma tabela no nosso template com os dados de todos os funcionários:

```
<table>
  <tbody>
    {% 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 %}
  </tbody>
</table>
```

Mais adiante, vamos falar sobre templates!!

As Views do Django

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

1. CreateView:Para criar objetos (É o Create do CRUD)
2. DetailView:Traz os detalhes do objeto (É o Retrieve do CRUD)
3. UpdateView:Para atualizar um objeto (É o Update do CRUD)
4. DeleteView:Para excluir um objeto (É o Delete do CRUD)

E várias outras muito úteis!!!

Agora vamos tratar detalhes do tratamento de requisições através de funções. Em seguida, mais sobre as CBVs.

Function Based View – Tratamento de Requisições

Utilizar funções é a maneira mais explícita para tratar requisições no Django. 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 é POST
    if request.method == 'POST' :
        form = FormularioDeCriacao(request.POST)
        if form.is_valid() :
            form.save()
            return HttpResponseRedirect(reverse('lista_funcionarios'))
    # qualquer outro método: GET, OPTION, DELETE, etc...
    else :
        return render(request, "templates/form.html", {'form' : form})
```

Primeiro, verificamos o método HTTP da requisição no campo method do objeto request

(linha 3)

Depois instanciamos um form com os dados da requisição (no caso POST) com FormularioDeCriacao(request.POST) (linha 4)

Verificamos os campos do formulário com o form.is_valid() (linha 5)

Se tudo estiver correto usamos o helper reverse() para traduzir a rota 'lista_funcionarios' para 'funcionarios'. Utilizamos isso para retornar um redirect para a view de listagem (linha 7)

Se for qualquer outro método, apenas renderizamos a página novamente com o método render() (linha 10)

Deu para perceber que o objeto request é essencial nas nossas Views, né?, separei aqui alguns atributos desse objeto que provavelmente serão os mais utilizados por você:

1. request.scheme:String representando o esquema (se veio por uma conexão HTTP ou HTTPS)
2. request.path:String com o caminho da página requisitada - exemplo: /cursos/curso-de-python/detalhes
3. request.method:Conforme citamos, contém o método HTTP da requisição (GET, POST, UPDATE, OPTION, etc)
4. request.content_type:Representa o tipo MIME da requisição - ex: (-text/plain para texto plano, image/png para arquivos PNG, etc)
5. request.GET:Um dict contendo os parâmetros GET da requisição
6. request.POST:Um dict contendo os parâmetros do corpo de uma requisição POST
7. 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"

8. request.COOKIES:Dict contendo todos os COOKIES no formato de string

Classes (CBV – Class Based Views) – Tratamento de Requisições

Conforme expliquei anteriormente, 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 a página inicial, seja mostrado apenas uma página simples, com as opções possíveis.
- Queremos que nossa página de listagem contenha a lista de todos os funcionários cadastrados no banco de dados.
- Queremos uma página com um formulário contendo todos os campos pré-preenchidos para atualização de dado do funcionário.
- Queremos uma página de exclusão de funcionários.
- Queremos um formulário em branco para a inclusão de um novo funcionário.

certo???

Pois é, a CBVs facilitam isso para nós!!! Temos basicamente duas formas para utilizar um CBV:

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

```
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 herdar da View desejada e sobrescrever os atributos e métodos na subclasse. Mais adiante veremos as Views mais utilizadas, e como podemos utilizá-las no nosso projeto.

TemplateView

Por exemplo, para o primeiro caso, podemos utilizar a TemplateView para renderizar uma página da seguinte forma (views.py):

```
from django.views.generic import TemplateView

class IndexTemplateView(TemplateView) :
```

```
    template_name = "index.html"
```

E a configuração de rotas ficam assim (urls.py):

```
from website.views import IndexTemplateView

urlpatterns = [
    path('', IndexTemplateView.as_view(), name='index'),

]
```

ListView

Para o segundo caso, de listagem de funcionários, podemos utilizar a ListView. Nela nós configuramos o Model que deve ser buscado (Funcionario no nosso caso), e ela automaticamente faz a busca por todos os registros presentes na tabela do banco de dados. Por exemplo (view.py):

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

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

Utilizamos o atributo context_object_name para nomear a variável que estará disponível no contexto do template (se não, o nome padrão dado pelo Django será object). -E a configuração de rotas fica assim:

```
from website.views import FuncionarioListView

urlpatterns = [
    #GET
    path('', views.index, name='index'),
    path('', IndexTemplateView.as_view(), name='index'),

    path('funcionarios/',
            FuncionarioListView.as_view(),
        name='lista_funcionarios'),
]
```

Isso resultará em uma página lista.html contendo um objeto chamado funcionarios contendo todos os Funcionários disponíveis para iteração. **DICA**: É uma boa prática colocar o nome da View como o Model + CBV base. Por exemplo: uma view que lista todos os Cursos, receberia o nome de CursoListView (Model=Curso e CBV = ListView).

Melhorando a Aparência da Listagem

No arquivo lista.html, vamos criar um CSS:

```
<style>
  table{
    width: 700px;
    text-align: center;
  }
  table thead{
    background-color: aquamarine;
    font-weight: bold;
  }
  table tr#campos{
    background-color: wheat;
    font-weight: bold;
  }
</style>
```

E o componente <table> vai receber alguns atributos para aplicar a formatação da CSS:

```
<table>
  <thead>
    <tr>
      <td colspan="4">Lista de Clientes</td>
    </tr>
  </thead>
  <tbody>
    <tr id="campos">
      <td>Nome</td>
      <td>Sobrenome</td>
      <td>Tempo de Serviço (Ano)</td>
      <td>Remuneração (R$)</td>
    </tr>
    {% for funcionario in funcionarios %}
    <tr>
      <td>{{ funcionario.nome }}</td>
      <td>{{ funcionario.sobrenome }}</td>
      <td>{{ funcionario.tempo_de_servico }}</td>
```

```
        <td>{{ funcionario.remuneracao }}</td>
      </tr>
      {% endfor %}
    </tbody>
</table>
```

UpdateView

Para a atualização de usuários podemos utilizar a UpdateView. Com ela, configuramos qual o Model (atributo model), quais campos (atributo field) e qual o nome do template (atributo template_name), e com isso temos um formulário para a atualização do modelo definido. No nosso caso:

```
from django.views.generic import UpdateView

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

DICA: Ao invés de listar todos os campos em fields em formato de lista de strings, podemos utilizar fields = '__all__'. Dessa forma, o Django irá buscar todos os campos para você!

Mas 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 dessa forma:

Na configuração de rotas(urls.py):

```
from website.views import FuncionarioUpdateView

urlpatterns = [
    # utilizando o {id}/{slug} para buscar o objeto
    path(
        'funcionario/<pk>',
        FuncionarioUpdateView.as_view(),
        name='atualiza_funcionario'),
]
```

Mas o que é slug?

Slug é uma forma de gerar URLs mais legíveis a partir de dados já existentes. Exemplo: podemos criar um campo slug utilizando o campo nome do funcionário. Dessa forma, as URLs ficariam assim:

/funcionario/vinicius-ramos

E não assim:

/funcionario/175

No campo slug, todos os caracteres são tranformados em minúsculos e os espaços são transformados em hífens, o que dá mais sentido à URL.

O arquivo atualiza.html vai renderizar o form de edição com o seguinte código:

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

  {{ form }}
  <button>Atualizar</button>
</form>
```

Dessa forma, os dados dos funcionários estarão disponíveis no formulário do template atualiza.html

DeleteView

Para deletar funcionários, utilizamos a DeleteView. Sua configuração é similar à UpdateView: nós devemos informar ao Django qual objeto queremos excluir via URLConf (urls.py) ou através do método get_object()

Precisamos configurar:

- O template que será renderizado
- O model associado à essa view
- O nome do objeto que estará disponível no template
- A URL de retorno, caso haja sucesso na exclusão

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

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

class FuncionarioDeleteView(DeleteView) :
  template_name = "website/exclui.html"
  model = Funcionario
```

```
    context_object_name = 'funcionario'
    success_url = reverse_lazy('website:lista_funcionarios')
```

O método reverse_lazy() serve para fazer a conversão de rotas (similar ao reverse()) mas em um momento em que o URLConf ainda não foi carregado (que é o caso aqui).

Assim como na UpdateView, fazemos a configuração do id a ser buscado no URLConf (urls.py), da seguinte forma:

```
from website.views import FuncionarioDeleteView

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

Assim, precisamos apenas fazer um template de confirmação da exclusão do funcionário (o link será feito através de um botão "Excluir" que vamos adicionar à página lista.html). Podemos fazer o template da seguinte forma:

```
<form method="post">
    {% csrf_token %}
    Você tem certeza que quer excluir o funcionário
    <b>{{ funcionario.nome }} {{funcionario.sobrenome}}</b>?
    <br/><br/>
    <a href="{% url 'website:lista_funcionarios' %}">
        <button type="button">Cancelar</button></a>
    <button>Excluir</button>
</form>
```

Algumas observações:

A tag do Django {% csrf_token %} 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)

Não se preocupe com a sintaxe do template, veremos mais sobre ele no próximo tópico.

CreateView

Nesta View, precisamos apenas dizer para o Django o model, o nome do template, a classe do formulário (vamos tratar mais sobre Forms adiante) e a URL de retorno, caso haja sucesso na inclusão do Funcionário. Podemos fazer isso assim:

```
from django.views.generic import CreateView

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

O método 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 website.views import FuncionarioCreateView

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

Com isso, estará disponível no template configurado (website/cria.html) um objeto form contendo os campos do 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 estilo, conforme o Django nos entrega, da seguinte forma:

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

OBSERVAÇÃO: Apesar de ser um Form, sua renderização não contém as tags <form></form> - cabendo a nós incluí-las no template.

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">
  {% csrf_token %}

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

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

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

  <label for="{{ form.tempo_de_servico.id_for_label }}">
    Tempo de Serviço
  </label>
  {{ form.tempo_de_servico }}

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

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

Nesse template:

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

Vamos aprender mais adiante sobre a utilização do Form do Django!

Capítulo 3

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, em código Python, os formulários das páginas HTML, simplificando e automatizando seu processo de criação e 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 seguinte formulário:

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

E que, ao ser submetido ao servidor, tenha seus campos de entrada validados e, em caso de validação positiva - sem erros, seja inserido no banco de dados. No centro desse sistema de formulário do Django está a classe Form. Nela descrevemos os campos que estarão disponíveis no formulário HTML. Para o formulário acima, podemos descrever assim:

```
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 descreve o tamanho máximo que esse input pode receber (100 caracteres)

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, optei por mostrar o essencial para começarmos a utilizá-lo. Vamos ver agora um exemplo mais complexo com um formulário de inserção de um Funcionário com todos os campos. Para isso, tenha criado o arquivo forms.py no app website:

```
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.CharField(
        required=True
    )
    remuneracao = forms.DecimalField()
```

Afff, mas o Model e o Form são quase iguais... Terei que reescrever os campos toda vez? Claro que não!!! Por isso o Django nos presenteou como incrível ModelForm. Com o ModelForm nós configuramos de qual Model o Django deve pegar os campos. A partir do atributo fields, nós dizemos quais campos nós queremos e, através do campo exclude, os campos que não queremos.

Para fazer essa configuração, utilizamos os metadados da classe interna Meta. Metadado (no caso do Model e do Form) é tudo aquilo que não será transformado em campo, como model, fields, ordering, etc. Assim, nosso ModelForm, pode ser descrito da seguinte forma:

```
from django import forms
from django.forms import ModelForm
from helloworld.models import Funcionario

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'
        ]
```

OBS: No arquivo views.py, importar o form e remover o comentário da linha:

```
from .forms import InsereFuncionarioForm

#form_class = InsereFuncionarioModel
```

Podemos utilizar apenas o campo fields, apenas o exclude ou os dois juntos e mesmo ao utilizá-los, 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 um exemplo:

```
    # Outros campos
    chefe = forms.BooleanField(
        label='Chefe?',
        required=True
    )

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

Isso vai gerar um formulário com:

- Todos os campos contidos em fields
- Serão retirados os campos contidos em exclude
- O campo forms.BooleanField, como um checkbox (<input type='checkbox' name='chefe' ...>)
- Biografia como 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 configurando o atributo widget com forms.TextArea, será renderizado como um textArea no nosso template.

Mais adiante, estudaremos um pouco mais sobre formulário, quando formos renderizá-los nos nossos templates.

Middleware

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 o arquivo settings.py, nós temos a lista de MIDDLEWARE com diversos middlewares pré-configurados. Por exemplo, temos o middleware AuthenticationMiddleware, ele é responsável por adicionar a variável user a todas as requisições. Assim, por exemplo, você pode mostrar o usuário logado no seu template:

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

Você pode pesquisar e perceber que em lugar nenhum em nosso código nós adicionamos a variável user ao Contexto das requisições. Não é muito comum, mas pode ser que você tenha que adicionar algum comportamento antes de começar a tratar a Requisição ou depois de formar a Resposta. Portanto, veremos 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 e, assim como uma View, pode ser escrito como Função ou como Classe. Um exemplo escrito como função é:

```
def middleware_simples(get_response):
    # Código de inicialização do Middleware
    def middleware(request):
        # Código a ser executado antes da View e
        # antes de outros middlewares serem executados
        response = get_response(request)
        # Código a ser executado após a execução
        # da View que irá processar a requisição
        return response
    return middleware
```

E como classe é:

```
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 antes da View e
        # antes de outros middlewares serem executados
        response = self.get_response(request)
        # Código a ser executado após a execução
        # da View que irá processar a requisição
        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:

O Método **process_view**, (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 será 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.

O método **process_exception**, (process_exception(request, exception))

Esse método é chamado 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 middleware de template, retornando a requisição ao browser

- request é o objeto HttpRequest
- exception é a exceção propriamente dita lançada pela view

O método **process_template_response**, (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 completo 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

Esse middleware é muito útil quando temos, por exemplo, um conjunto de servidores com IP fixo que vão se conectar entre si. Você poderia, por exemplo, ter uma configuração no seu settings.py chamada ALLOWED_SERVERS contendo a lista de IP autorizados a se conectar ao seu serviço

Para isso, precisamos abrir o cabeçalho das requisições que chegam no nosso servidor e verificar se o IP de origem está autorizado. Como precisamos dessa lógica antes da requisição chegar na View, vamos adicioná-la ao método process_view, da seguinte forma:

Crie um arquivo chamado middlewares.py dentro do diretório helloworld:

```
from django.http import Http404, HttpResponse, HttpResponseForbidden

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 usuário 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 = [
    '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.FiltraIPMiddleware',
]
```

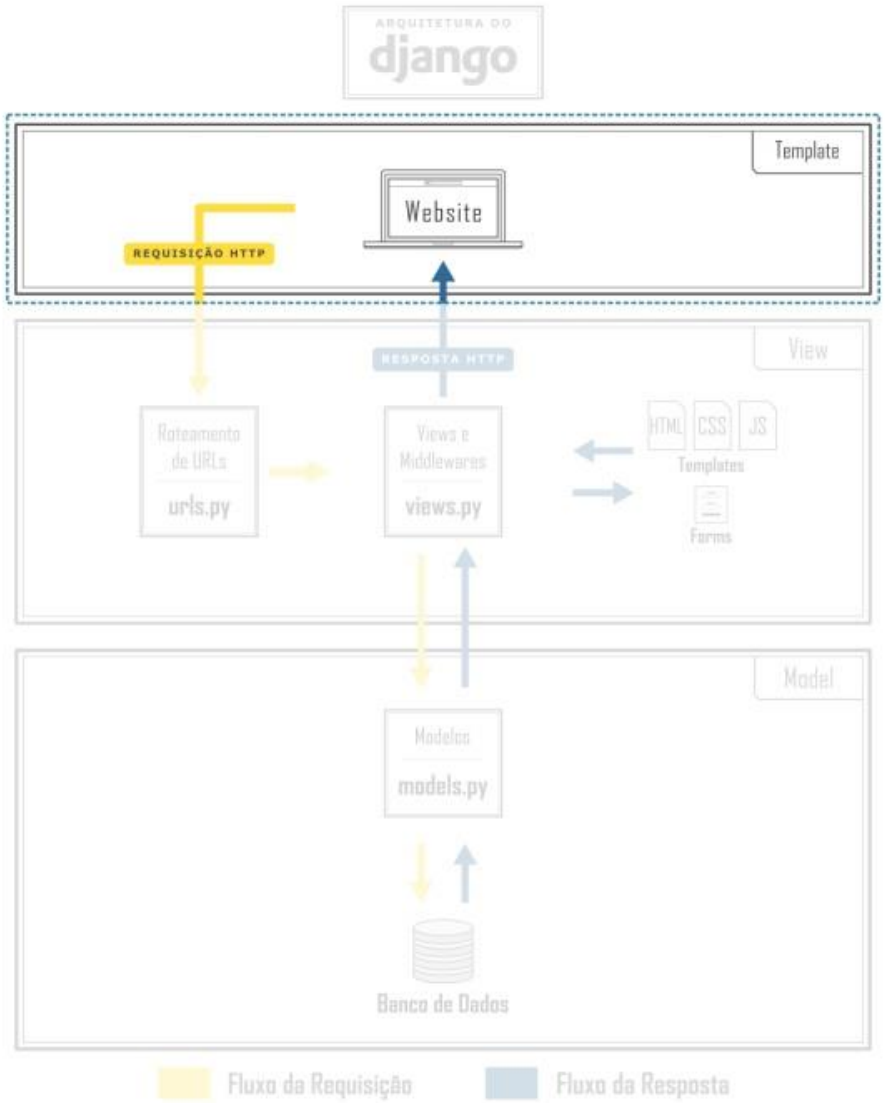
Agora, podemos testar seu funcionamento alterando a lista ips_autorizados:

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 "IP não autorizado", pois nosso IP não está autorizado a acessar o servidor

A Camada Template

Chegamos ao nosso último capítulo, vamos aprender a configurar, customizar e estender templates, como utilizar os filtros e tags do Django, como criar tags e filtros customizados e um pouquinho de Bootstrap para deixar as páginas bonitonas!!!



A Camada Template é quem dá cara à nossa aplicação, isto é, faz a interface com o usuário. 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!

Agora, 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, mas, afinal, o que é um 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 no Django contém:

- Variáveis que podem ser substituídas por valores, a partir do processamento por uma Engine de Templates (núcleo ou "motor" de templates). Usamos os marcadores {{ variável }}
- Tags que controlam a lógica do template. Usamos com {% tag %}
- Filtros que adicionam funcionalidades ao template. Usamos com {{ variável | filtro }}

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

```
1  {# base.html contém o template que usaremos como esqueleto #}
2  {% extends "base.html" %}
3
4  {% block conteudo %}
5   <h1>{{ section.title }}</h1>
6
7   {% for f in funcionarios %}
8    <h2>
9     <a href="{% url 'website:funcionario_detalhe' pk=f.id %}">
10     {{ funcionario.nome|upper }}
11    </a>
12   </h2>
13  {% endfor %}
14 {% endblock %}
```

Alguns pontos importantes:

Linha 1: Escrevemos comentário com a tag {# comentário #}

Linha 2: Utilizamos {% extends "base.html" %} para estender de um template, ou seja, utilizá-lo como base, passando o caminho para ele

Linha 4: Podemos facilitar a organização do template, criando blocos com {% block nome_do_bloco %} {% endblock %}

Linha 5: Podemos interpolar variáveis vindas do servidor com nosso template com {{ secao.titulo }} - dessa forma, estamos acessando o atributo titulo do objeto secao (que deve estar no Contexto da resposta)

Linha 7: É possível iterar sobre objetos de uma lista através da tag {% for objeto in lista %} {% endfor %}

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: {{ funcionario.nome|upper|cut:" " }}

Para facilitar a manipulação de templates, os desenvolvedores do Django criaram uma linguagem que contém todos esses elementos. Chamaram-na de DTL – Django Template Language! Veremos mais dela nesse capítulo! Para começarmos a utilizar os templates do Django, é necessário primeiro configurar sua utilização, e é isso que veremos agora:

(Template Django) Configuração

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

```
1 TEMPLATES = [
2   {
3     'BACKEND': 'django.template.backends.django.DjangoTemplates',
4     'DIRS': [],
5     'APP_DIRS': True,
6     'OPTIONS': {},
7    },
8 ]
```

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 pelos 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 parâmetros em OPTIONS

Por ora, vamos utilizar as configurações padrão "de fábrica" pois elas já nos atendem! Agora, vamos ver sobre o tal Django Template Language!

Django Template Language (DTL)

A DTL é a linguagem padrão dos templates do Django. Ela é simples, porém poderosa. Dando uma olhada na sua documentação, podemos entender a filosofia da DTL. 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 (p.ex: PHP)

Isto é o terror dos designers (e não só deles)! Ponha-se no lugar de um designer que não sabe nada de 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 DTL do que botando a mão na massa e melhorando as páginas da nossa aplicação, né?! OBS.: Neste projeto vamos utilizar o Bootstrap 4, para dar um tapa no visual

Template-base

Nosso template que servirá de esqueleto deve conter o código HTML que se repetirá 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 no site Bootstrap e jQuery, que são dependências necessárias para os nossos templates. Baixe os arquivos e coloque-os na pasta website/static/ do projeto

Agora, com os devidos arquivos colocados na pasta, podemos começar com o nosso template_base:

```
<!DOCTYPE html>
<html>
{% load static %}
<head>
 <title>
  {% block title %}Gerenciador de Funcionários{% endblock %}
 </title>

 <!-- Estilos -->
 <link rel="shortcut icon" type="image/png"
  href="{% static 'website/img/favicon.png' %}">
 <link rel="stylesheet"
  href="{% static 'website/css/bootstrap.min.css' %}">
```

```
 <link rel="stylesheet"
  href="{% static 'website/css/master.css' %}">

 {% block styles %}{% endblock %}
</head>

<body>
 <nav class="navbar navbar-expand-lg navbar-light">
  <a class="navbar-brand" href="{% url 'website:index' %}">
   <img src="{% static 'website/img/favicon.png' %}" height='45px'>
  </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="{% url 'website:index' %}">
      Página Inicial
     </a>
    </li>
    <li class="nav-item">
     <a class="nav-link" href="{% url 'website:lista_funcionario' %}">
      Funcionários
     </a>
    </li>
   </ul>
  </div>
 </nav>

 {% block conteudo %}{% endblock %}

 <script src="{% static 'website/js/jquery.min.js' %}"></script>
 <script src="{% static 'website/js/bootstrap.min.js' %}"></script>

 {% block scripts %}{% endblock %}

 <script src="{% static 'website/js/scripts.js' %}"></script>
</body>
</html>
```

E pronto! Temos um template base! Agora vamos customizar a tela principal da nossa aplicação: a index.html!

Página Inicial (Template: website/index.html)

No diretório website, crie o arquivo 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 seguinte tag do Django, que serve para que um template estenda de outro: {% extends "caminho/para/template" %}

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

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

```

```
<!-- Bloco de conteúdo da nossa página -->
{% block conteudo %}
<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="{% url 'website:cadastra_funcionario' %}"
      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="{% url 'website:lista_funcionarios' %}"
      class="btn btn-primary">
      Vá para Lista
     </a>
    </div>
   </div>
  </div>
 </div>
</div>
{% endblock %}
```

Com isso, nossa página inicial - ou nossa Homepage - fica estilizada, antes, vamos verificar os arquivos views.py e urls.py novamente:

VIEWS.PY

```
from django.urls import reverse_lazy
from django.views.generic import TemplateView, ListView, UpdateView, CreateView, DeleteView
from helloworld.models import Funcionario
from website.forms import InsereFuncionarioForm

# PÁGINA PRINCIPAL
class IndexTemplateView(TemplateView) :
    template_name = "website/index.html"

#LISTA DE FUNCIONÁRIOS
class FuncionarioListView(ListView) :
    template_name = "website/lista.html"
    model = Funcionario
    context_object_name = "funcionarios"

#ATUALIZAÇÃO DE FUNCIONÁRIOS
class FuncionarioUpdateView(UpdateView) :
    template_name = "website/atualiza.html"
    model = Funcionario
    fields = '__all__'
    context_object_name = 'funcionario'
    success_url = reverse_lazy('website:lista_funcionarios')

#EXCLUSÃO DE FUNCIONÁRIOS
class FuncionarioDeleteView(DeleteView) :
    template_name = "website/exclui.html"
    model = Funcionario
    context_object_name = 'funcionario'
    success_url = reverse_lazy('website:lista_funcionarios')

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

URLs.PY

```
from django.urls import path
from website.views import IndexTemplateView, FuncionarioListView,
FuncionarioUpdateView, FuncionarioCreateView, FuncionarioDeleteView

app_name = 'website'

# urlpatterns contém a lista de roteamento de URLs
urlpatterns = [
   # GET /
   path('', IndexTemplateView.as_view(), name='index'),

   # GET /funcionarios
   path('funcionarios/', FuncionarioListView.as_view(), name='lista_funcionarios'),

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

   #GET/POST /funcionario/excluir/{pk}
   path('funcionario/excluir/<pk>', FuncionarioDeleteView.as_view(),
name='deleta_funcionario'),

   # GET /funcionario/cadastrar/
   path('funcionario/cadastrar/', FuncionarioCreateView.as_view(),
name='cadastra_funcionario'),
]
```

Cadastro de Funcionários (Template:website/cria.html)

Nesse template, mostramos o formulário para cadastro de novos funcionários. Se lembra que definimos o formulário InsereFuncionarioForm? Vamos utilizá-lo nesse template, adicionando-o na View FuncionarioCreateView. Dessa forma, ela irá expor um objeto form no nosso template para que possamos utilizá-lo

Mas, antes de seguir, vamos instalar uma biblioteca que vai nos auxiliar e muito a renderizar os campos de input do nosso formulário: a Widget Tweaks! Com ela, nós temos maior liberdade de 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: {% load widget_tweaks %}

E pronto, agora podemos utilizar a tag que irá renderizar os campos do formulário, a render_field: {% render_field nome_do_campo parametros %}

Para alterar como o input será renderizado, utilizamos os parâmetros da tag. Dessa forma, podemos alterar o código HTML resultante. Portanto, nosso template pode ser escrito assim:

```
{% extends "website/_layouts/base.html" %}

{% load widget_tweaks %}

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

{% block conteudo %}
<div class="container">
 <div class="row">
  <div
   class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
   <div class="card">
```

```
13      <div class="card-body">
14       <h5 class="card-title">Cadastro de Funcionário</h5>
15       <p class="card-text">
16        Complete o formulário abaixo para cadastrar
17        um novo <code>Funcionário</code>.
18       </p>
19       <form method="post">
20        <!-- Não se esqueça dessa tag -->
21        {% csrf_token %}
22
23        <!-- Nome -->
24        <div class="input-group mb-3">
25         <div class="input-group-prepend">
26          <span class="input-group-text">Nome</span>
27         </div>
28         {% render_field form.nome class+="form-control" %}
29        </div>
30
31        <!-- Sobrenome -->
32        <div class="input-group mb-3">
33         <div class="input-group-prepend">
34          <span class="input-group-text">Sobrenome</span>
35         </div>
36         {% render_field form.sobrenome class+="form-control" %}
37        </div>
38
39        <!-- CPF -->
40        <div class="input-group mb-3">
41         <div class="input-group-prepend">
42          <span class="input-group-text">CPF</span>
43         </div>
44         {% render_field form.cpf class+="form-control" %}
45        </div>
46
47        <!-- Tempo de Serviço -->
48        <div class="input-group mb-3">
49         <div class="input-group-prepend">
50          <span class="input-group-text">
51           Tempo de Serviço
52          </span>
53         </div>
54         {% render_field form.tempo_de_servico class+="form-control" %}
55        </div>
```

```
56
57          <!-- Remuneração -->
58          <div class="input-group mb-3">
59           <div class="input-group-prepend">
60            <span class="input-group-text">Remuneração</span>
61           </div>
62           {% render_field form.remuneracao class+="form-control" %}
63          </div>
64
65          <button class="btn btn-primary">Enviar</button>
66         </form>
67        </div>
68       </div>
69      </div>
70     </div>
71    </div>
72    {% endblock %}
```

Com isso, o formulário fica renderizado com estilo bastante agradável e limpo!

Listagem de Funcionários (Template:website/lista.html)

Nesta página, nós queremos mostrar o conjunto de Funcionários cadastrados no banco de dados e as ações que o usuário pode tomar: atualizar os dados do 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:

```
1  {% extends "website/_layouts/base.html" %}
2
3  {% block title %}Lista de Funcionários{% endblock %}
4
5  {% block conteudo %}
6  <div class="container">
7   <div class="row">
8    <div class="col-lg-12 col-md-12 col-sm-12 col-xs-12">
9     <div class="card">
10     <div class="card-body">
11      <h5 class="card-title">Lista de Funcionário</h5>
12
13      {% if funcionarios|length > 0 %}
14      <p class="card-text">
15       Aqui está a lista de <code>Funcionários</code>
16       cadastrados.
17      </p>
```

```
18      <table class="table">
19       <thead class="thead-dark">
20        <tr>
21         <th>ID</th>
22         <th>Nome</th>
23         <th>Sobrenome</th>
24         <th>Tempo de Serviço</th>
25         <th>Remuneração</th>
26         <th>Ações</th>
27        </tr>
28       </thead>
29
30       <tbody>
31        {% for f in funcionarios %}
32         <tr>
33          <td>{{ f.id }}</td>
34          <td>{{ f.nome }}</td>
35          <td>{{ f.sobrenome }}</td>
36          <td>{{ f.tempo_de_servico }}</td>
37          <td>{{ f.remuneracao }}</td>
38          <td>
39           <a href="{% url 'website:atualiza_funcionario' pk=f.id %}"
40            class="btn btn-info">
41            Atualizar
42           </a>
43           <a href="{% url 'website:deleta_funcionario' pk=f.id %}"
44             class="btn btn-outline-danger">
45             Excluir
46            </a>
47           </td>
48          </tr>
49         {% endfor %}
50        </tbody>
51       </table>
52      {% else %}
53       <div class="text-center mt-5 mb-5 jumbotron">
54        <h5>Nenhum <code>Funcionário</code> cadastrado ainda.</h5>
55       </div>
56      {% endif %}
        <hr />
        <div class="text-right">
```

```
57          <a class="btn btn-primary"
58          href="{% url 'website:cadastra_funcionario' %}">
59           Cadastrar Funcionário
60          </a>
61         </div>
62        </div>
63       </div>
64      </div>
65    </div>
66    {% endblock %}
```

Você terá um resultado, sem funcionários cadastrados terá uma aparência e com funcionário cadastrado terá outra. 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

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 que implementamos nos capítulos anteriores. Ela expõe um formulário com os campos preenchidos com os dados atuais para que o usuário possa alterar

Vamos utilizar novamente a biblioteca Widget Tweaks para facilitar a renderização dos campos input. Vamos construir o nosso template assim:

```
1  {% extends "website/_layouts/base.html" %}
2
3  {% load widget_tweaks %}
4
5  {% block title %}Atualização de Funcionário{% endblock %}
6
7  {% block conteudo %}
8  <div class="container">
9
```

```
<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 -->
     {% csrf_token %}

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

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

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

```

```
53          {% render_field form.tempo_de_servico class+="form-contro
54  l" %}
55        </div>
56
57        <!-- Remuneração -->
58        <div class="input-group mb-3">
59         <div class="input-group-prepend">
60          <span class="input-group-text">Remuneração</span>
61         </div>
62         {% render_field form.remuneracao class+="form-control" %}
63        </div>
64        <button class="btn btn-primary">Enviar</button>
65       </form>
66      </div>
67     </div>
68    </div>
69   </div>
70  </div>
71 </div>
72 {% endblock %}
```

Perceba que o seu código é semelhante ao template de adição de Funcionários, com os campos sendo renderizados com a tag render_field. Como nossa View herda de UpdateView, o objeto form já vem populado com os dados do modelo em questão (aquele cujo id foi enviado ao se clicar no botão de edição)

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. Essa página vai concretizar a sua exclusão. A view que fizemos, a FuncionarioDeleteView, facilita bastante nossa vida. Com ela, basta dispararmos uma requisição POST para a URL configurada, que o Funcionário será deletado! Dessa forma, nosso objetivo se resume à:

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

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

<!-- Bloco de conteúdo da nossa página -->
{% block conteudo %}
 <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">
     {% csrf_token %}
     <hr />
     <div class="text-right">
      <a href="{% url 'website:lista_funcionarios' %}"
       class="btn btn-outline-danger">
       Cancelar
      </a>
     <button class="btn btn-danger">Excluir</button>
    </div>
   </form>
  </div>
 </div>
</div>
{% endblock %}
```

Nada de inédito no código apresentado acima. Pronto! Com isso, temos todas as páginas do nosso projeto! Agora vamos ver como construir tags e filtros customizados!

Tags e Filtros Customizados

Sabemos, até 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 necessários.

Por isso, ele previu a possibilidade de você construir seus próprios filtros e tags! Portanto, 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

Os filtros e tags customizados residem em uma pasta específica da nossa estrutura: a /templatetags. Portanto, crie na raiz do app website essa pasta (website/templatetags) e adicione:

- Um script __init__ em braço (para que o Django enxergue como um pacote Python)
- O script tempo_atual.py em branco referente à nossa tag
- O script primeira_letra.py em branco referente ao nosso filtro

Nossa estrutura, portanto, 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 instalados esteja configurada na lista INSTALLED_APPS do settings.py (no nosso caso, website já está lá, portanto, nada a fazer aqui).

Também é necessário carrega-los com o {% load filtro/tag %}, vamos chama-lo de primeira_letra e, quando estiver pronto, iremos utilizá-lo da seguinte maneira:

```
<p>{{ valor|primeira_letra }}</p>
```

Filtro primeira_letra

Filtros customizados são basicamente 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 {{ valor | primeira_letra }} :

- valor será o value
- Nosso filtro não irá receber argumentos, portanto não foi passado nada para ele

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 as tags e filtros são registrados). Isso define um filtro!

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 a definição de diversos filtros que podemos utilizar em nossos templates (eu separei alguns e vou explicar logo abaixo):

Lá temos o exemplo do filtro lower:

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

Nele:

- @register.filter(is_safe=True) é um decorator 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.

Com isso, vamos agora codificar e registrar o nosso filtro! Uma forma de pegarmos a primeira letra de uma string é transfomá-la em lista e pegar o elemento de índice [0], da seguinte forma:

```
1 from django import template
2 from django.template.defaultfilters import stringfilter
3
4 register = template.Library()
5
6 @register.filter
7 @stringfilter
8 def primeira_letra(value):
9   return list(value)[0]
```

Nesse código:

- O código register = template.Library() é necessário para pegarmos uma instância da biblioteca filtros do Django. Com ela, podemos registrar nosso filtro com @register.filter
- @register.filter e @stringfilter são os decorators que citei logo acima

E agora vamos testar, fazendo o carregamento e utilização em algum template. Para isso, vamos alterar a tabela do template website/lista.html para incluir nosso filtro da seguinte forma:

```
<!-- Primeiro, carregamos nosso filtro, logo após o extends -->
{% load primeira_letra %}
...
<table class="table">
 <thead class="thead-dark">
  <tr>
   <th><!-- Retiramos o "ID" aqui --></th>
   <th>Nome</th>
   <th>Sobrenome</th>
   <th>Tempo de Serviço</th>
   <th>Remuneração</th>
   <th class="text-center">Ações</th>
  </tr>
 </thead>
 <tbody>
 {% for f in funcionarios %}
  <tr>
   <!-- Aplicamos nosso filtro no atributo funcionario.nome -->
   <td>{{ f.nome|primeira_letra }}</td>
   <td>{{ f.nome }}</td>
   <td>{{ f.sobrenome }}</td>
   <td>{{ f.tempo_de_servico }}</td>
   <td>{{ f.remuneracao }}</td>
   <td class="text-center">
    <a class="btn btn-primary"
```

```
    href="{% url 'website:atualiza_funcionario' pk=f.id %}">
     Atualizar
    </a>
    <a class="btn btn-danger"
     href="{% url 'website:deleta_funcionario' pk=f.id %}">
     Excluir
    </a>
   </td>
  </tr>
 {% endfor %}
 </tbody>
</table>
```

E com isso, terminamos 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 podemos 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. Assim como na criação da tag, precisamos incluir uma instância de template.Library (para ter acesso à biblioteca de filtros e tags do Django), utilizar o decorator @register (para registrar nossa tag) e definir a implementação da nossa função.

Para pegar o tempo atual, podemos utilizar o método now() da biblioteca datetime. Como queremos formatar a data, também utilizamos o método strftime(), passando como parâmetro a string formatada (%H é a hora, %M são os minutos e %S são os segundos).

Podemos, então, definir nossa tag da seguinte forma:

```
1 import datetime
2 from django import template
3
4 register = template.Library()
5
6 @register.simple_tag
7 def tempo_atual():
8     return datetime.datetime.now().strftime('%H:%M:%S')
```

E para utilizá-la, a carregamos com {% load tempo_atual %} e a utilizamos em nosso template com {% tempo_atual %}. No nosso caso, vamos utilizar nossa tag no template-base: o website/_layouts/base.html.

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="{% url 'website:index' %}">
      Página Inicial
     </a>
    </li>
```

```
    <li class="nav-item">
     <a class="nav-link" href="{% url 'website:lista_funcionarios' %}">
      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>{% tempo_atual %}
     </a>
    </li>
   </ul>
  </div>
 </nav>
  ...
```

Com isso, temos nosso filtro e tag customizados! Agora vamos dar uma olhada nos 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 considero mais úteis! Sem mais delongas, aí vai o primeiro: o capfirst!!!

Filtro capfirst

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

Exemplo: valor = 'esse é um texto'.

Utilização:

```
{{ valor|capfirst }}
```

Saída:

```
Esse é um texto
```

Filtro cut

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

Exemplo: valor = 'Esse É Um Texto De Testes'.

Utilização:

```
{{ valor|cut:" " }}
```

Saída:

```
EsseÉUmTextoDeTestes
```

Filtro date

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

Exemplo: Entrada: Objeto datetime.

Utilização:

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

Saída:

```
01/07/2018
```

Filtro default

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

Exemplo: Entrada: valor = False

Utilização:

```
{{ valor|default:'Nenhum valor' }}
```

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:

```
{{ valor|default:'Nenhum valor' }}
```

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:

```
{{ valor|divisibleby:'2' }}
```

Saída:

```
True
```

Filtro filesizeformat

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

Exemplo: Entrada: valor = 123456789

Utilização:

```
{{ valor|filesizeformat }}
```

Saída:

```
117.7 MB
```

Filtro first

O que faz: Retorna o primeiro item em uma lista.

Exemplo: Entrada: valor = [“Marcos”, “João”, “Luiz”]

Utilização:

```
{{ valor|first }}
```

Saída:

```
Marcos
```

Filtro last

O que faz: Retorna o último item em uma lista.

Exemplo: Entrada: valor = [“Marcos”, “João”, “Luiz”]

Utilização:

```
{{ valor|last }}
```

Saída:

```
Luiz
```

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:

```
{{ valor|floatformat:"2" }}
```

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:

```
{{ valor|join:" - " }}
```

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 (se length > 0, lista não está vazia).

Exemplo: Entrada: valor = [“Marcos”, “João”]

Utilização:

```
1 {% if valor|length > 0 %}
2 <p>Lista contém valores</p>
3 {% else %}
4 <p>Lista vazia</p>
5 {% endif %}
```

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:

```
1 {{ valor|lower }}
```

Saída:

```
paralelepípedo
```

Filtro pluralize

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

Exemplo: Entrada: valor = 12

Utilização:

```
1 Sua empresa tem {{ valor }} Funcionário{{ valor|pluralize:"s" }}
```

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, 8, 9]

Utilização:

```
1 {{ valor|random }}
```

Saída:

Sua saída será um valor da lista escolhido randomicamente.

Filtro title

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

Exemplo: Entrada: valor = 'primeiro post do blog'

Utilização:

```
1 {{ valor|title }}
```

Saída:

```
Primeiro Post Do Blog
```

Filtro upper

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

Exemplo: Entrada: valor = 'texto de testes'

Utilização:

```
1 {{ valor|upper }}
```

Saída:

```
TEXTO DE TESTES
```

Filtro wordcount

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

Exemplo: Entrada: valor = 'Django é o melhor framework da web'

Utilização:

```
1 {{ valor|wordcount }}
```

Saída:

```
6
```

Conclusão

Nesse capítulo 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, para deixar as páginas bonitonas!

Para lhe ajudar, vou colocar aqui algumas referências para você se manter atualizado e também para aprender cada vez mais sobre o Django:

Site oficial do Django: https://www.djangoproject.com/
Documentação: https://docs.djangoproject.com/pt-br/2.0/
Grupo do Facebook: https://www.facebook.com/groups/django.brasil/