**CONTINUAÇÃO DJANGO...**

**Configurações importantes para o Django:**

Retirando informações confidenciais das settings com a biblioteca python-decouple:

pip install python-decouple

Após isso, criar um arquivo com nome .env na raiz do projeto e adicionar as informações sigilosas. Segue um exemplo com a informação SECRET\_KEY, do arquivo settings.py:

Texto

Descrição gerada automaticamente

Adicionando em .env:

Tela de computador com fundo preto

Descrição gerada automaticamente

Após isso, importando config da biblioteca decouple e trocando o valor de SECRET\_KEY para a variável criada em .env:

Texto

Descrição gerada automaticamente

Outro exemplo de informação que deve ser inserida é o **DEBUG**, que fica da seguinte maneira:

Texto

Descrição gerada automaticamente

O valor ‘cast’ presente no config transforma o valor recebido, que por padrão vem como string, para booleano (ou seja, sempre que o valor recebido pelo config precisar ser de um tipo que não seja string, será preciso utilizar o cast para converter ao tipo desejado). E o valor default defini um valor padrão, nesse caso como False, caso não exista um valor definido para DEBUG no arquivo .env

Mais um exemplo diferente é no caso de ALLOWED\_HOSTS, que recebe uma lista de valores. No arquivo .env, não podemos inserir uma lista, então apenas inserimos os valores separados por vírgula e utilizamos a função Csv para converter isso em uma lista dentro de settings.py.

Segue o exemplo:

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente com confiança média

\*Aqui foi utilizado o \* para dizer que é aceito qualquer tipo de host, mas num projeto real será os hosts permitidos separados por vírgula

Dentro de settings.py:

Texto

Descrição gerada automaticamente

**ENVIANDO VALORES DA VIEW PARA OS TEMPLATES:**

Para enviar valores das views para os templates, utilizamos o contexto que é enviado no render. Segue um exemplo onde temos a varável nome e alunos definidas e enviamos para o template todo\_list.html:

Texto

Descrição gerada automaticamente

Os valores do contexto são enviados no terceiro parâmetro de render, dentro de uma chave, conforme a imagem acima. O nome da chave é o nome que será chamado nos templates para acessar o valor, que recebe a variável criada com os valores.

Para acessar o valor enviado no template é utilizado a seguinte sintaxe:

Texto

Descrição gerada automaticamente

\*O nome da chave enviada vai dentro de dois colchetes ‘{{ }}’, podendo ser iterada utilizando o for de Django, conforme a imagem ilustra.

**PUXANDO AS INFORMAÇÕES DO BANCO DE DADOS PARA AS VIEWS:**

Para puxar as informações do banco de dados para as views precisamos primeiramente importar o nosso Model criado em models. Após isso, criamos uma variável para receber o(s) valor(es) que queremos e chamamos a função objects para obter esses dados do model. Veja um exemplo:

Texto

Descrição gerada automaticamente

\***IMPORTANTE**: todo model criado possui a função objects, que é utilizada para manipular os dados do banco de dados (model), nesse caso para colher essa informação

Nesse caso, pegamos todas as informações do model Todo, usando a função ‘all()’.

Para exibir esses dados em template, fazemos da mesma forma que a anterior. Segue exemplo:

Texto

Descrição gerada automaticamente

Nesse caso estamos pegando a coluna ‘title’ de cada cadastro do banco de dados todo. Exibição no site:

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Exibindo o title ‘Estudar Django’ (até então, único cadastro do banco de dados realizado)

Segue um exemplo da lista de tarefas com estilização usando bootstramp:

Interface gráfica do usuário

Descrição gerada automaticamente com confiança baixa

O template todo\_list ficou da seguinte forma:

Texto

Descrição gerada automaticamente

\*As setas indicam o que foi utilizado do django-html para adicionar as informações

**COMEÇANDO A TRABAHAR COM CLASS BASED VIEW**

A partir daqui, começaremos a trabalhar com class based views no lugar de function based views, pois é um modelo mais recente e recomendado para trabalhar com as views de django.

Para começar, a primeira mudança que precisa ser feita nas views é a seguinte:

Texto

Descrição gerada automaticamente

**IMPORTANTE!!!**

Por padrão, a ListView vai procurar uma template do app que tenha o mesmo nome do modelo (Todo no nosso caso) + \_list, ou seja, todo\_list. Como já criamos a template seguindo esse padrão, funcionou tudo sem precisar ficar alterando várias coisas. A única alteração que foi feita foi a seguinte:

Foi importado a generic view ListView e usando herança, criamos a classe TodoListView e indicamos com qual modelo a classe deve trabalhar (Todo, nesse caso) criando a variável model.

Realizando isso, precisamos alterar o arquivo urls.py também, já que mudamos a função para uma classe. Segue as alterações a serem feitas:

Tela preta com letras brancas

Descrição gerada automaticamente

**CRIANDO UMA CREATEVIEW**

Para que o usuário consiga criar tarefas, precisamos de um novo view que possibilite a criação de um formulário. Para criar esse view com class based view, utilizamos o CreateView:

Texto

Descrição gerada automaticamente

Ao criar essa classe de view, declaramos o model. apontando para o model que vamos utilizar e o fields, que será uma lista com os campos que o cliente irá preencher no formulário.

Após isso, devemos nos lembrar de adicionar a nova view nas urlpatterns, lá em urls.py:

Texto

Descrição gerada automaticamente

Realizando esses passos, também iremos precisar criar uma template para a nossa view create, então criamos com o nome todo\_forms.html. Para adicionarmos os campos que informamos em fields dentro do template, implementamos o forms.

Segue uma imagem de como implementar o forms:

Texto

Descrição gerada automaticamente

Criamos um simples formulário html com um botão de submit e adicionamos o forms no duplo colchete, o que gera o seguinte resultado:

Interface gráfica do usuário

Descrição gerada automaticamente

Mas, se tentarmos enviar o formulário dessa forma, ocorrerá um erro:

Uma imagem contendo Gráfico

Descrição gerada automaticamente

Isso é uma medida de segurança trazida pelo próprio Django.

Para garantir que não tenhamos esse erro, precisamos adicionar a template tag {% csrf\_token %} no seu form, conforme o print:

Texto

Descrição gerada automaticamente

Porém, se tentarmos enviar o formulário agora, encontraremos um novo erro:

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Esse erro acontece porque não definimos para onde o usuário será redirecionado em caso de sucesso de envio do formulário. Para definirmos isso, precisamos seguir alguns passos:

Nomear as urls criadas em urlpatterns:

Texto

Descrição gerada automaticamente

Após isso, nas views do app, começamos importando de django.urls o reverse\_lazy, uma função que faz a partir do nome da url puxa todo o seu path. Feito isso, adicionamos a variável succes\_url na createview, chamando o reverse\_lazy e passando o nome da url que será apontada quando houver sucesso no envio do formulário. Segue imagem ilustrando:

Texto

Descrição gerada automaticamente

Realizando essas ações, o formulário passará a funcionar e conseguiremos enviá-lo, sendo mandado para a listagem de tarefas ao termos sucesso no envio do formulário:

Interface gráfica do usuário

Descrição gerada automaticamente com confiança baixa

**CRIANDO BOTÃO PARA IR A TELA DE CADASTRO**

Para criarmos um botão na lista de tarefas que redirecione para a tela de cadastro é bem simples, precisamos simplesmente utilizar a template-tag url:

Tela de computador com texto preto sobre fundo escuro

Descrição gerada automaticamente

A template tag url faz um trabalho parecido com o reserve-lazy, localizando o caminho da url apenas pelo seu nome. Assim, criamos um botão que redireciona para a tela de cadastro:

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

**REALIZANDO ESTILIZAÇÃO DO FORMULÁRIO COM DJANGO CRISPY FORMS**

Nas aulas, o professor utilizou uma biblioteca do django para estilização dos formulários, para não ficar criando CSS durante o conteúdo de django. Para utilizar essa biblioteca, é preciso importa-la:

pip install crispy-bootstrap5

Depois adicionamos algumas coisas no settings.py:

Primeiramente, para melhor organização dos APPS, criamos três categorias: DJANGO\_APPS, THIRDY\_PARTY\_APPS e MY\_APPS. Em DJANGO\_APPS adicionamos os apps que vem por padrão no django. Em THIRDY\_PARTY\_APPS, apps que adicionamos de terceiros. E em MY\_APPS, os apps que criamos.

Tendo feito isso, adicionamos as seguintes apps em THIRDY\_PARTY:

THIRDY\_PARTY\_APPS = [

    'crispy\_forms',

    'crispy\_bootstrap5',

]

Segue como ficou os APPS após isso:

# Application definition

DJANGO\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

]

THIRDY\_PARTY\_APPS = [

    'crispy\_forms',

    'crispy\_bootstrap5',

]

MY\_APPS = ['todos.apps.TodosConfig',]

INSTALLED\_APPS = DJANGO\_APPS + THIRDY\_PARTY\_APPS + MY\_APPS

E por último, adicionar as seguintes configurações no final do arquivo setting.py:

CRISPY\_ALLOWED\_TEMPLATE\_PACKS = "bootstrap5"

CRISPY\_TEMPLATE\_PACK = "bootstrap5"

Agora, para usar a biblioteca, adicionamos as seguintes informações no template todo\_forms.py:

No ínicio do arquivo (antes do DOCTYPE):

{% load crispy\_forms\_tags %}

E a tag form vai ficar da seguinte forma:

<main *class*="container">

        <h1>Nova tarefa</h1>

        <form *method*="POST">

            {% csrf\_token %}

            {{ form|crispy }}

            <button *type*="submit" *class*="btn btn-primary">Salvar</button>

        </form>

    </main>

E para que os campos fiquem em português no site, adicionamos um verbose name para as colunas diretamente no models, da seguinte forma:

Texto

Descrição gerada automaticamente

Assim obtemos esse resultado:

Forma, Retângulo

Descrição gerada automaticamente

**EXTENÇÃO DE TEMPLATES – HERANÇA DE TEMPLATES**

Isso é algo muito importante para otimização e um ponto muito positivo do Django. A extensão de templates permite que criamos uma espécie de herança de template, onde criamos um template base com conteudo que é comum entre vários outros templates e somente extendemos esse template base, para que não seja necessário refazer o mesmo conteúdo várias vezes.

Então ná partica, usando o nosso todo\_list e todo\_form, somente duas coisas variam entre eles: o título e o main. Então podemos fazer da seguinte forma:

criamos um base.html que recebe todo o conteúdo que se repete:

<!DOCTYPE html>

<html *lang*="pt-br">

  <head>

    <meta *charset*="utf-8">

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1">

    <title>TWTodos - {% block page\_title %}{% endblock page\_title %}</title>

    <link *href*="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" *rel*="stylesheet" *integrity*="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" *crossorigin*="anonymous">

  </head>

  <body>

    <!-- As a link -->

    <nav *class*="navbar bg-dark mb-4" *data-bs-theme*="dark">

        <div *class*="container-fluid">

        <a *class*="navbar-brand" *href*="#">TWTodos</a>

        </div>

    </nav>

    <main *class*="container">

        {% block content %}{% endblock content %}

    </main>

    <script *src*="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" *integrity*="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" *crossorigin*="anonymous"></script>

  </body>

</html>

**CRIANDO VIEW PARA ATUALIZAÇÃO DOS CADASTROS**

Agora, iremos criar uma view para a atualização dos cadastros já feitos. O primeiro passo é importar a classe ViewUpdate e criar uma nova class view, herdando dessa classe. Segue o exemplo:

Texto

Descrição gerada automaticamente

Até aqui a criação é quase idêntica a criação da CreateView, mas mudará quando formos adicionar a view nas urls.py. Como para atualizar precisamos primeiramente localizar um cadastro, a UpdateView precisa ser inserida da seguinte forma em urls.py:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

\*\***IMPORTANTE**: perceba que primeiro importamos a view update que criamos e depois adicionamos um novo path, inserindo um nome para url (update) e depois inserindo a variável do id do cadastro.

Para inserirmos váriaveis assim no django, utilizamos o <>, informando primeiramente o tipo do dado da variável e depois o nome da váriavel (nesse caso utilizamos o pk, que é a chave primaria do cadastro, ou seja, o id).

A UpdateView puxa o mesmo arquivo por padrão que o CreateView, já que as ações são semelhantes. Porém, para mudarmos alguns nomes como título da página ou algum texto podemos utilizar o if do django no template, conforme o exemplo:

Texto

Descrição gerada automaticamente

\*Nesse caso usamos o if todo.pk, pois só existirá um pk caso estejamos em um cadastro já criado. Se entrarmos para criar um novo cadastro, todo.pk será false e entraremos no else, aparecendo as opções de novo cadastro.

CRIANDO UM BOTÃO DE EDITAR NA TODO\_LIST

Após criarmos a view de update, devemos criar um botão para que o usuário possa editar a tarefa que deseja. Para isso, precisamos adicionar um novo botão e colocar a url do UpdateView, conforme o print:

Texto

Descrição gerada automaticamente

\*Note que diferente da url\_create que criamos anteriormente, essa precisa de uma váriavel para funcionar, já que em urls criamos ela dessa forma.

No final, a página inicial do todo\_list ficará assim:

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**CRIANDO VIEW PARA EXCLUSÃO DE CADASTROS**

Seguindo o mesmo raciocínio das views anteriores, iremos criar uma delete view, importanto a generic view DeleteView, conforme o print abaixo:

Texto

Descrição gerada automaticamente

Ao criar a view, devemos seguir o mesmo passo das views anteriores e adicioná-la na url.py, conforme apresentado no print abaixo:

Texto

Descrição gerada automaticamente

\*Da mesma forma que o update recebe a variável pk, o delete também receberá.

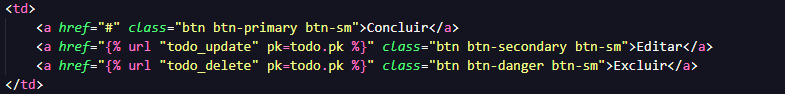
Porém, diferentemente do update, o delte view precisá de um novo template com o nome *app*\_confirm\_delete, ou no nosso caso, todo\_confirm\_delete. Esse template serve para que haja uma confirmação da exclusão desse cliente e precisa ser criado para não ocorrer erros no sistema.

Segue um exemplo de como o template fica:

Texto

Descrição gerada automaticamente

E não podemos também esquecer de adicionar o caminho da URL no botão excluir do todo\_list, conforme o print:



**CRIANDO A VIEW PARA FINALIZAR AS TAREFAS**

Diferente das outras views que criamos, a view para completar as tarefas será criado de outra forma, pois é uma ação mais específica ao modelo do nosso site e não tem uma view genérica do django que faça isso. Nesse caso, iremos criar a classe TodoCompleteView e herdar da classe View, a classe genérica para class based views.

Dentro da classe, precisaremos realizar as configurações manualmente. A primeira que criamos, será a função de get e para isso utilizaremos o shortcut django get\_object\_or\_404, que retornará o que foi solicitado ou um erro HTTP 404.

A função get\_object\_or\_404 recebe dois parâmetros, o models que será utilizado e o parâmetro da busca que será feita. Segue um print de como deve ser feito:

Texto

Descrição gerada automaticamente

\*Nesse caso, estamos pegando o valor da tarefa com o get\_object\_or\_404, depois atrelando ao campo finished\_at dessa tarefa a data de hoje com a biblioteca date. Após isso, para salvar as alterações, utilizamos o todo.save() e redirecionamos para a todo\_list com a função redirect. Segue um print das importações feitas:

Texto

Descrição gerada automaticamente

Após essa configuração, precisamos realizar a configuração da view em urls.py, que não difere das anteriores que foram feitas. Segue o print:

Tela preta com letras brancas

Descrição gerada automaticamente

E também devemos realizar a configuração na todo\_list, no botão completar:

Texto

Descrição gerada automaticamente

**CRIANDO CLASS META E ORDENANDO POR DATA DE ENTREGA:**

Fazendo alguns ajustes no site, é interessante colocar as tarefas por ordem de data de entrega e uma forma de realizarmos isso é utilizando a class meta dos models django. Dentro dessa class Meta podemos realizar algumas configurações, como por exemplo, alterar a ordem de exebição dos itens. Segue o exemplo do print:

Texto

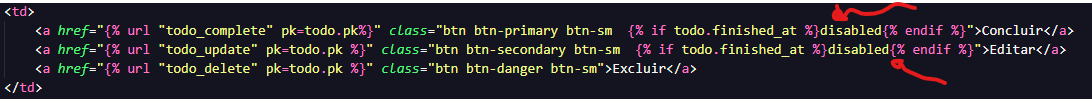
Descrição gerada automaticamente

\*Nesse caso, ordenamos utilizando como parêmetro a coluna deadline.

OBS: para deixar a ordem em ordem decrescente, é só colocar um “-“ antes da palavra, nesse caso ficaria: ordering = [‘-deadline’]

**DESABILITANDO BOTÕES AO CONLUIR TAREFA:**

Outra mudança para melhora do site é a criação de if para verificar se a tarefa já foi concluída. Em caso positivo, será desativado tanto o botão de concluir quanto o de editar. Segue print:



Como fica no site:

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

**MELHORIA NO CÓDIGO:**

Aqui é uma mudança apenas estrutural, não afetando o site em sí. Em django, há uma filosofia que diz que é melhor ter models mais cheios e view mais vazias, por isso é sempre bom mantermos a regra de négocio dentro dos models e somente chamar ela nas views. Sege um exemplo:

Texto

Descrição gerada automaticamente

\*Em models, criamos a função mark\_has\_complete para settar o finished\_at para um valor caso ele não tenha um valor. E nas views, apenas chamamos a função que criamos:

Texto

Descrição gerada automaticamente

Dessa forma, podemos sempre que precisarmos chamar a função que conclui a tarefa, sem precisa repetir em outros lugares toda a lógica, além também de precisar alterar somente em um local.