Passo-a-passo para a criação de uma aplicação Django

Tutorial baseado nos cursos da Alura

# Criação de uma VENV

Na pasta onde vamos criar nossa aplicação, através do prompt de comando:

c:\>python -m venv .\NOME\_DA\_VENV

ou

c:\>python -m venv c:\path\to\NOME\_DA\_VENV

\* Normalmente o nome da venv acaba sendo venv.

## Ativando a venv

No prompt de comando:

.\venv\Scripts\activate

# Instalação do Django

Com a venv ativada.

\* se estiver usando o VS CODE, "ctrl + Shift + P" e digitar ">python select interpreter" e selecionar o python com a venv

pip install django==2.2.6

# Iniciando o projeto

django-admin startproject NOME\_DO\_PROJETO .

Usa-se o ponto por conta de arrumação de pastas mais simples

No caso o comando será:

django-admin startproject alurareceita .

## Vendo os arquivos

No arquivo settings.py inicialmente mudar:

LANGUAGE\_CODE = 'pt-br'

TIME\_ZONE = 'America/Sao\_Paulo'

# Rodar o servidor

Para confirmar se a instalação ocorreu bem. Esse é o “Olá mundo” do Django.

python manage.py runserver

# Começando a aplicação

python manage.py startapp NOME\_DO\_APP

No nosso caso:

python manage.py startapp receitas

## Registrando o app

No arquivo settings.py, incluir no INSTALLED\_APPS o app criado.

INSTALLED\_APPS = [

'NOME\_DO\_APP',

(...)

]

No caso será 'receitas'

INSTALLED\_APPS = [

'receitas',

(...)

]

## Acessando o app

CRIAR um arquivo "urls.py" no app:

from django.urls import path

from . import views

e incluir:

urlpatterns = [

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

]

No arquivo de "views.py" criar

from django.http import HttpResponse

def index(request):

return HttpResponse('<h1>Receitas</h1>')

Voltar para urls.py:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('', include('receitas.urls')),

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

]

# Criando os templates

Dentro da pasta "receitas" (nome do app), criar a pasta "templates" e dentro desta pasta, criar o arquivo: index.html e dentro da views.py corrigir o conteúdo:

Apagando o from django.http import HttpResponse e deixando:

from django.shortcuts import render

def index(request):

return render(request, 'index.html')

acessando o link pegamos o front-end preparado

<https://caelum-online-public.s3.amazonaws.com/1489-django+parte+1/site_receitas.zip>

# Arquivos estáticos

No arquivo "setting.py":

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [os.path.join(BASE\_DIR, 'receitas/templates')],

//código omitido

},

]

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

STATIC\_URL = '/static/'

STATICFILES\_DIRS = [

os.path.join(BASE\_DIR, 'alurareceita/static')

]

Criar dentro da pasta do projeto, "alurareceita" uma pasta 'static' e dentro desta pasta criar as pastas:

css, fonts, img, js, scss

no caso vamos colar as pastas existentes que baixamos no link

python manage.py collectstatic

## Carregando os arquivos estáticos

Na primeira linha dos arquivos html digitar o código:

**{% load static %}**

Nas referências do html os endereços relativos tem que ter código python

<!-- Favicon -->

<link rel="icon" href="{% static 'img/core-img/favicon.ico' %}”>

<!-- Stylesheet -->

<link rel="stylesheet" href="{% static 'site.css' %}">

Procurar em todo o html as “src=” para colocar o controle de estático

# Links, urls e views

Primeiramente vamos referenciar o caminho do index com o código:

{% url 'index' %}

O código ficará assim:

<a class="nav-brand" href="{% url 'index' %}">

Agora adicionar a segunda página.

No arquivo urls.py da pasta receitas, adicionar o segundo path, ficando:

urlpatterns = [

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

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

]

Em seguida, no arquivo views.py fazer a segunda rota:

def index(request):

return render(request, 'index.html')

def receita(request):

return render(request, 'receita.html')

Agora no Código html, no início, colocar o código:

{% load static %}

E substituir os links de receitas para a url

<a href="{% url 'receita' %}">

# Estendendo html

Método para a criação de um modelo base de html e estender para todos os outros:

Criar um arquivo “base.html” e colocar a parte que se repete nas páginas: header e footer.

Na parte que vai entrar o código das páginas vai ser limitado com código:

{% block content %} {% endblock %}

Nas páginas html que vão estender a base vão começar com o código:

{% extends 'base.html'%}

{% load static %}

{% block content %}

No final:

{% endblock %}

## Partials

Forma que é possível compartilhar um trecho de html nos demais arquivos.

Primeiro criar uma pasta “partials” dentro da pasta de templates.

Depois criar arquivos html com as partes desejadas, no caso concreto, vamos fazer um menu, lembrando que os arquivos html devem ter como início

{% load static %}

No arquivo que vai usar colocamos o código:

{% include 'partials/menu.html' %}

Da mesma maneira que foi feito com o menu, foi feito com o footer:

{% include 'partials/footer.html' %}

# preenchimento dinâmico de conteúdo (Nomes de receitas dinâmicas)

Método para repetir uma estrutura com pequenas alterações de forma dinâmica;

Como exemplo, na página index é apresentado uma lista de receitas.

No arquivo de “views”, no retorno do render usa-se um terceiro parâmetro (contexto) na forma de dicionário.

A variável criada no arquivo html é colocado entre duas chaves.

E a parte que se repete estará dentro de um laço for:

No caso concreto:

{% for chave, nome\_da\_receita in nome\_das\_receitas.items %}

<!-- Single Best Receipe Area -->

<div class="col-12 col-sm-16 col-lg-4">

<div class="single-best-receipe-area mb-30">

<img src="{% static 'img/gb-img/foto\_receita.png' %}></img>

<div class="receipe-content">

<a href="receita.html">

<h5>{{ nome\_da\_receita }}</h5>

</a>

</div>

</div>

</div>

{% endfor %}

De maneira preliminar, vamos usar um dicionário na função index na views.

def index(request):

receitas = {

1:'Lasanha',

2:'Sopa de legumes',

3:'Sorvete'

}

dados = {

'nome\_das\_receitas' : receitas

}

return render(request, 'index.html',dados)

# Banco de dados

Para esse projeto será usado o banco de dados PostgreSQL. A escolha é da versão 10.xxx pois tem suporte para todos do sistemas operacionais.

A instalação vai ser a default.

## Configuração preliminar PostgreSQL

Depois de instalado o PostgreSQL, vamos executar o pgAdmin4 e criar um servidor para usar.

1 clicando com o direito no elefante pede-se para criar um servidor

2. atribuímos um nome

3. na aba conexão, nesse caso, usaremos localhost

4. criação de senha: no caso vamos usar !Q@W#E$R%T

5. clicando com o direto na palavra databases vamos criar um database, no caso, alura\_receita

Agora que temos um servidor habilitado vamos ao Django:

Instalar o módulos:

pip install psycopg2

pip install psycopg2-binary

no arquivo Settings.py vamos configurar o banco

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'alura\_receita',

'USER': 'postgres',

'PASSWORD': '!Q@W#E$R%T',

'HOST': 'localhost'

}

}

## Usando o banco de dados

Na pasta do app (receitas) vamos usar o arquivo models.py:

Inicialmente, No caso do projeto vai ficar assim:

from django.db import models

from datetime import datetime

class Receita(models.Model):

nome\_receita = models.CharField(max\_length

=200)

ingredientes = models.TextField()

modo\_preparo = models.TextField()

tempo\_preparo = models.IntegerField()

rendimento = models.CharField(max\_length

=100)

categoria = models.CharField(max\_length

=100)

date\_receita = models.DateTimeField(default=datetime.now, blank=True)

Depois de criados os modelos das tabelas, vamos fazer a migração (criação dos esquemas/tabelas):

python manage.py makemigrations

em seguida:

python manage.py migrate

# Django Admin

No arquivo admin.py na pasta do app( receitas) vamos importar os models e registrar os modelos. No caso concreto:

from django.contrib import admin

from .models import Receita

admin.site.register(Receita)

Criação de um administrador:

python manage.py createsuperuser

Criamos um perfil de administrador e senha:

No caso concreto:

Usuário: dudu1626

Senha: !Q@W#E$R%T

Agora é possível fazer o CRUD completo do site pelo perfil de administrador pelo endereço:

http://127.0.0.1:8000/admin/

## Exibindo dados dos banco

No arquivo views.py importamos os models e disponibilizamos para a leitura:

from django.shortcuts import render

from .models import Receita

def index(request):

receitas = Receita.objects.all()

dados = {

'receitas' : receitas

}

return render(request, 'index.html', dados)

def receita(request):

return render(request, 'receita.html')

Caso alguma mensagem de erro apareça, provavelmente será solucionado instalando o Pylint do Django

pip install pylint-django

se estiver no VS code, colocar o seguinte código na configuração:

"python.pythonPath": "venv/bin/python",

"python.linting.pylintArgs": [

"--load-plugins=pylint\_django"

],

Para o html apresentar os dados, vamos alterar o código, inicialmente checando se há dados:

{% if receitas %}

{% for receita in receitas %}

<!-- Single Best Receipe Area -->

<div class="col-12 col-sm-16 col-lg-4">

<div class="single-best-receipe-area mb-30">

<img src="{% static 'img/gb-img/foto\_receita.png' %}></img>

<div class="receipe-content">

<a href="receita.html">

<h5>{{ receita.nome\_receita }}</h5>

</a>

</div>

</div>

</div>

{% endfor %}

{% else %}

{% endif %}

## Parâmetro na url

Para visualizar os dados que estão no banco em uma tela:

Para passar um item do banco de dados vamos ter que passar o id do banco:

1. No HTML:

Configurar passando a id

<a href="{% url 'receita' receita.id %}">

1. No urls.py:

Configurar o caminho recebendo a id

urlpatterns = [

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

path('<int:receita\_id>', views.receita, name='receita'),

]

1. No views.py

Configurar que vai receber a id:

from django.shortcuts import render, get\_list\_or\_404, get\_object\_or\_404

def receita(request, receita\_id):

receita = get\_object\_or\_404(Receita, pk=receita\_id)

receita\_a\_exibir = {

'receita' : receita

}

return render(request, 'receita.html', receita\_a\_exibir)

Para tirar o erro, vamos tirar a navbar de receitas no htm do menu.

Por ultimo configurar o html para exibir os dados:

<!-- Receipe Content Area -->

<div class="receipe-content-area">

<div class="container">

<div class="row">

<div class="col-12 col-md-8">

<div class"receipe-headline my-5">

<span>{{ receita.date\_receita }}</span>

<h2>{{ receita.nome\_receita }}</h2>

<div class="receipe-duration">

<h6>Preparo: {{ receita.tempo\_preparo }} minutos</h6>

<h6>Rendimento: {{ receita.rendimento }}</h6>

<h6>Categoria: {{ receita.categoria }}</h6>

<h6>Por: Pessoa</h6>

</div>

</div>

</div>

<div class="row">

<div class="col-12 col-lg-18">

<!-- Single Preparation Step -->

<div class="single-preparation-step d-flex">

<p>{{ receita.modo\_preparo }}</p>

</div>

</div>

<!-- Ingredientes -->

,div class="col-12 col-lg-4">

<div class="ingredients">

<h4>Ingredientes</h4>

<div class="ingredients">

<p>{{ receita.ingredientes }}</p>

</div>

</div>

//código omitido

</div>

# Melhorias na página do admin

## Listando receitas por nome

A preocupação desta aula é melhorar a apresentação das receitas no painel do admin.

No arquivo admin.py vamos fazer a configuração para apresentar como lista e quais informações serão exibidas.

from django.contrib import admin

from .models import Receita

class ListandoReceitas(admin.ModelAdmin):

list\_display = ('id', 'nome\_receita', 'categoria', 'tempo\_preparo')

list\_display\_links = ('id', 'nome\_receita')

admin.site.register(receita, ListandoReceitas)

list\_display\_links = ('id', 'nome\_receita') determina quais serão links para poder selecionar.

list\_display determina quais dos atributos serão mostrados.

## Busca, Filtros e paginação

Para ativar um campo para busca usamos search\_fields = ('nome\_receita',), atentar que esse argumento somente aceita tuplas, com uma simples virgula se resolve.

Para criar um filtro usamos list\_filter = ('categoria',), no caso queremos filtrar por categoria somente.

Para determinar quantas receitas vão aparecer por página usamos list\_per\_page = 2.

O Código da classe ficará dessa maneira:

from django.contrib import admin

from .models import Receita

class ListandoReceitas(admin.ModelAdmin):

list\_display = ('id', 'nome\_receita', 'categoria', 'tempo\_preparo')

list\_display\_links = ('id', 'nome\_receita')

search\_fields = ('nome\_receita',)

list\_filter = ('categoria',)

list\_per\_page = 2

admin.site.register(Receita, ListandoReceitas)

# Criando um modelo de pessoas

Nesse capítulo vamos criar um segundo app dentro do nosso projeto

1. criar um app com o comando:

python manage.py startapp pessoas

2. cadastrar o app no arquivo settings.py do projeto:

INSTALLED\_APPS = [

'receitas',

'pessoas',

// Código omitido

3. no arquivo models.py do app criar a classe:

from django.db import models

class Pessoa(models.Model):

nome = models.CharField(max\_length=200)

email = models.CharField(max\_length=200)

4. no arquivo admin.py do app registrar a pessoa:

from django.contrib import admin

from .models import Pessoa

admin.site.register(Pessoa)

5. no prompt fazer o comando de montar a migração:

python manage.py makemigrations

6. no prompt fazer a migração para o banco de dados:

python manage.py migrate

7. no arquivo admin.py do app de pessoas fazer as personalizações que fizemos em receitas:

from django.contrib import admin

from .models import Pessoa

class ListandoPessoas(admin.ModelAdmin):

list\_display = ('id', 'nome', 'email')

list\_display\_links = ('id', 'nome')

search\_fields = ('nome',)

list\_per\_page = 2

admin.site.register(Pessoa, ListandoPessoas)

# Integrando modelos

Primeiramente, como no caso concreto temos alguns dados no banco e queremos a vinculação entre as tabelas, vamos deletar o conteúdo da tabela de receitas.

No models.py de receitas vamos importar a classe pessoas e depois criar um atributo de chave estrangeira:

from django.db import models

from datetime import datetime

from pessoas.models import Pessoa

class Receita(models.Model):

pessoa = models.ForeignKey(Pessoa, on\_delete=models.CASCADE)

nome\_receita = models.CharField(max\_length=200)

ingredientes = models.TextField()

modo\_preparo = models.TextField()

tempo\_preparo = models.IntegerField()

rendimento = models.CharField(max\_length=100)

categoria = models.CharField(max\_length=100)

date\_receita = models.DateTimeField(default=datetime.now, blank=True)

def \_\_str\_\_(self):

return self.nome\_receita

depois dessa alteração, fazer o procedimento de migração, mas atenção: como o banco já existia e foi apagado vai ter uma mensagem perguntado o que fazer com os dados na tabela e o que fazer para o tratamento:

no caso vamos escolher a opção 1 que é informar qual o padrão a ser considerado em seguida vamos dar como padrão ‘’ uma string vazia

python manage.py makemigrations

python manage.py migrate

Depois de criadas novas receitas, vamos alterar o html de receitas para colocar a variável pessoas:

{{ receita.pessoa }}

# Seleção de conteúdo a publicar (Filtro receitas publicadas)

Criaremos um atributo em receita do tipo booleano, onde se falso, ele não aparecerá no site.:

No arquivo models de receitas:

publicada = models.BooleanField(default=False)

Fazer o procedimento de de migração de banco:

python manage.py makemigrations

python manage.py migrate

Para ter efeito a flag de exibição, teremos que colocar uma regra no arquivo views.py:

antes receitas selecionava todos com o comando = Receita.objects.all

def index(request):

receitas = Receita.objects.filter(publicada=True)

dados = {

'receitas' : receitas

}

return render(request, 'index.html', dados)

# Ordenação e edição no admin

## edição

Se colocarmos a flag no arquivo admin.py do app receitas podemos ver, mas não editar na listagem. A solução é:

list\_editable = ('publicada',)

## ordenação

Se quiséssemos colocar a ordem de exibição as receitas mais modernas na frente, a solução é Colocar um filtro no views:

receitas = Receita.objects.order\_by('-date\_receita').filter(publicada=True)

## Foto para cada receita

Para vincular uma foto para cada item do banco é possível através da criação de um atributo específico, tendo atenção para o local/diretório de salvamento da foto:

foto\_receita = models.ImageField(upload\_to='fotos/%d/%m/%Y', blank=True)

Temos que criar uma rota para os arquivos de mídia, no arquivo settings, vamos adicionar:

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

MEDIA\_URL = '/media/'

Antes de fazer o procedimento de migração, devemos instalar um módulo para trabalhar com imagens:

pip install pillow

Fazer o procedimento de migração de banco:

python manage.py makemigrations

python manage.py migrate

## Exibindo a foto

Para conseguir exibir as fotos no site os procedimentos são:

1. Configurar o arquivo urls.py do projeto para receber mídias:

from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

path('', include('receitas.urls')),

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

] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

1. No arquivo html do index vamos colocar um cógigo para checar se tem uma imagem no banco ou não, se não tiver usamos uma imagem padrão:

ATENÇÃO QUE A VARIÁVEL DA FOTO TEM FINAL.URL

{% if receita.foto\_receita == ‘’ %}

<img src="{% static 'img/bg-img/foto\_receita.png' %}></img>

{% else %}

<img src="{{ receita.foto\_receita.url }}" alt="">

{% endif %}

# Criando a página de busca

Para estruturar uma busca no site devemos:

1. No arquivo de urls de receitas (app), vamos incluir o path de busca

from django.urls import path

from . import views

urlpatterns = [

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

path('<int:receita\_id>', views.receita, name='receita'),

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

]

1. No arquivo de views vamos criar uma função específica para isso:

def buscar(request):

lista\_receitas = Receita.objects.order\_by('-date\_receita').filter(publicada=True)

if 'search' in request.GET:

nome\_a\_buscar = request.GET['search ']

if buscar:

lista\_receitas = lista\_receitas.filter(nome\_receita\_\_icontains=nome\_a\_buscar)

dados = {

'receitas' : lista\_receitas

}

return render(request, 'buscar.html', dados)

1. Criar um arquivo html para estruturar a busca:

Inicialmente é uma cópia do html do index com uma resposta para quando a busca não encontrar receita.

(trecho)

{% if receitas %}

{% for receita in receitas %}

<!-- Single Best Receipe Area -->

<div class="col-12 col-sm-6 col-lg-4">

<div class="single-best-receipe-area mb-30">

{% if receita.foto\_receita == '' %}

<img src="{% static 'img/bg- img/foto\_receita.png' %}" alt="">

{% else %}

<img src="{{ receita.foto\_receita.url }}" alt="">

{% endif %}

<div class="receipe-content">

<a href="{% url 'receita' receita.id %}">

<h5>{{ receita.nome\_receita }}</h5>

</a>

</div>

</div>

</div>

{% endfor %}

{% else %}

<div class="col-12 col-sm-6 col-lg-4">

<h3>Receita não encontrada!</h3>

</div>

{% endif %}

1. Ajustar a url de busca no arquivo html que tem o link de busca:

NÃO PRECISA DO MÉTODO DE POST, O CÓDIGO JÁ TEM

<div class="col-12">

<form action="{% url 'buscar' %}">

<input type="search" name="search" placeholder="O que está procurando..."></input>

<button type="submit"><i class="fa fa-search" aria hidden="true"></i></button>

</form>

## Refatorando a busca

Vamos criar uma partial com o código de busca com o nome de busca.html:

{% load static %}

<!-- Search Wrapper -->

<div class="search-wrapper">

<!-- Close Btn -->

<div class="close-btn"><i class="fa fa-times" aria-hidden="true"></i></div>

<div class="container">

<div class="row">

<div class="col-12">

<form action="{% url 'buscar' %}">

<input type="search" name="search" placeholder="O que está procurando...">

<button type="submit"><i class="fa fa-search" aria-hidden="true"></i></button>

</form>

</div>

</div>

</div>

</div>

Nos arquivos html de index, receitas e buscar devemos incluir:

{% include 'partials/busca.html' %}

Uma boa prática é nomear os partials com um underline no início do nome. Por exemplo:

\_footer.html, \_menu.html, etc

# Autorização e usuários

Na parte de administração do Django o super usuário pode inserir usuários e selecionar os níveis de permissão para editar na parte de admin.

# Criando o app de usuários

Nesse capítulo vamos criar um terceiro app dentro do nosso projeto, usuários que irão se logar, incluir e alterar receitas.

1. criar um app com o comando:

python manage.py startapp usuarios

1. cadastrar o app no arquivo settings.py do projeto:

INSTALLED\_APPS = [

'receitas',

'usuarios',

'pessoas',

// Código omitido

1. Criar um arquivo urls.py ficando assim:

from django.urls import path

from . import views

urlpatterns = [

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

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

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

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

]

1. Criar views para todos os caminhos criados no urls:

Ver nos tópicos abaixo

1. No arquivo de urls do projeto também precisa ser atualizada com o caminho para o arquivo url de usuários:

from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

path('', include('receitas.urls')),

path('usuarios/', include('usuarios.urls')),

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

] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

1. Criar os templates html de login, nesse momento foi refatorado o local da pasta de templates, ajustando no arquivo de settings a configuração de templates:

\*No caso do curso os arquivos html foram cedidos

1. Não vamos precisar fazer o procedimento de models porque vamos usar o banco de dados do django (usado pelo admin).

## Requisições no Django

Nos arquivos de html que possuem formulários, devemos configurar o método do formulário para POST e na configuração de ação nós configuramos a url que vai receber o dado do formulário.

<form action="{% url 'cadastro' %}" method="POST">

E

<form action="{% url 'login' %}" method="POST">

## CSRF, token e dados

O Django tem um item de proteção obrigatória para formulários que é chamado de crsf token, utilizado colocando o código logo depois da tag de formulário POST

<form action="{% url 'cadastro' %}" method="POST">

{% csrf\_token %}

## Criando usuários

A obtenção das informações do formulário html é através do método request.POST[‘nome\_do\_item\_no\_html’] como o nosso caso:

nome = request.POST['nome']

email = request.POST['email']

senha = request.POST['password']

senha2 = request.POST['password2']

No curso optamos por cadastrar o usuário somente com essas informações, mas o banco de dados de usuários do Django contém como possibilidade os seguintes registros:

* password
* last\_login
* is\_superuser
* username
* first\_name
* last\_name
* email
* is\_staff
* is\_active
* date\_joined

por enquanto nosso view de cadastro será com o seguinte código e com os controles básicos

def cadastro(request):

if request.method == 'POST':

nome = request.POST['nome']

email = request.POST['email']

senha = request.POST['password']

senha2 = request.POST['password2']

if not nome.strip():

print('O campo nome não pode ficar em branco')

return redirect('cadastro')

if not email.strip():

print('O campo email não pode ficar em branco')

return redirect('cadastro')

if senha != senha2:

print('As senhas não são iguais')

return redirect('cadastro')

if User.objects.filter(email=email).exists():

print('Usuário já cadastrado')

return redirect('cadastro')

user = User.objects.create\_user(username=nome, email=email, password=senha)

user.save()

print('Usuário cadastrado com sucesso')

return redirect('login')

else:

return render(request,'usuarios/cadastro.html')

## Login e dashboard

O procedimento de configuração inicial do login é semelhante ao de cadastro, porém, se o login for feito com sucesso, o usuário será redirecionado para um dashboard:

from django.shortcuts import render, redirect

def login(request):

if request.method == 'POST':

email = request.POST['email']

senha = request.POST['password']

print(email, senha)

if email == "" or senha == "":

print('Os campos email e senha não podem ficar em branco')

return redirect('login')

return redirect('dashboard')

return render(request, 'usuarios/login.html')

Logo, será necessário configurar o dashboard e criar um html para ele. O código será basicamente o mesmo que o index, mas com um pequeno detalhe, depois do {% include 'partials/\_menu.html' %} incluir a parte de boas-vindas para o usuário

<div class="contact-area section-padding-0-80">

<div class="container">

<div class="row">

<div class="col-12">

<div class="section-heading">

<h3>Olá {{ user.username }}</h3>

</div>

</div>

</div>

</div>

</div>

No arquivo views:

def dashboard(request):

if request.user.is\_authenticated:

id = request.user.id

receitas = Receita.objects.order\_by('-date\_receita').filter(pessoa=id)

dados={

'receitas': receitas,

}

return render(request, 'usuarios/dashboard.html', dados)

else:

return redirect('index')

### Realizando o Login

Inicialmente devemos importar o pacote de autenticação:

from django.contrib import auth

O padrão de login do Django é com o username, mas no projeto o professor escolheu usar o email, então será feita uma gambiarra, se o email digitado existir, vai se buscar o username:

(Trecho omitido)

if User.objects.filter(email=email).exists():

nome = User.objects.filter(email=email).values\_list('username', flat=True).get()

(Trecho omitido)

A autenticação será :

if User.objects.filter(email=email).exists():

nome = User.objects.filter(email=email).values\_list('username', flat=True).get()

user = auth.authenticate(request, username=nome, password=senha)

if user is not None:

auth.login(request, user)

return redirect('dashboard')

Se estiver logado, as opções de menu de cadastro e login sumirão e a opção de logout aparecerá, para tanto, incluiremos uma validação se estiver logado ou não no html de menu:

Trecho omitido

<!-- Nav Start -->

<div class="classynav">

<ul>

{% if user.is\_authenticated %}

<li><a href="{% url 'index' %}">Home</a></li>

<li><a href="{% url 'dashboard' %}">Minhas Receitas</a></li>

<li><a href="{% url 'logout' %}">Logout</a></li>

{% else %}

<li><a href="{% url 'index' %}">Home</a></li>

<li><a href="{% url 'cadastro' %}">Cadastro</a></li>

<li><a href="{% url 'login' %}">Login</a></li>

{% endif %}

</ul>

Trecho omitido

### Realizando o logout

A parte de logout é a mais simples, tem um método para isso. Para ficar lógico o site redireciona para o index

def logout(request):

auth.logout(request)

return redirect('index')

## Criando formulário de receita

1. Criar um caminho no arquivo de url de usuários:

path('cria\_receita', views.cria\_receita, name='cria\_receita'),

1. Criar uma função em views para implementar

def cria\_receita(request):  
 if request.method == 'POST':  
 nome\_receita = request.POST['nome\_receita']  
 ingredientes = request.POST['ingredientes']  
 modo\_preparo = request.POST['modo\_preparo']  
 tempo\_preparo = request.POST['tempo\_preparo']  
 rendimento = request.POST['rendimento']  
 categoria = request.POST['categoria']  
 foto\_receita = request.FILES['foto\_receita']  
 user = get\_object\_or\_404(User, pk=request.user.id)  
 # maneira de pegar o usuário na requisição  
 receita = Receita.objects.create(pessoa=user, nome\_receita=nome\_receita,  
 ingredientes=ingredientes, modo\_preparo=modo\_preparo,  
 tempo\_preparo=tempo\_preparo, rendimento=rendimento,  
 categoria=categoria, foto\_receita=foto\_receita)  
 receita.save()  
 return redirect('dashboard')  
 else:  
 return render(request, 'usuarios/cria\_receita.html')

1. Criar um botão no menu para levar para a página de criação:

<div class="classynav">

<ul>

{% if user.is\_authenticated %}

<li><a href="{% url 'index' %}">Home</a></li>

<li><a href="{% url 'dashboard' %}">Minhas Receitas</a></li>

<li><a href="{% url 'cria\_receita' %}">Criar Receita</a></li>

<li><a href="{% url 'logout' %}">Logout</a></li>

{% else %}

<li><a href="{% url 'index' %}">Home</a></li>

<li><a href="{% url 'cadastro' %}">Cadastro</a></li>

<li><a href="{% url 'login' %}">Login</a></li>

{% endif %}

</ul>

1. Devemos corrigir o model anterior de receita que vinculava as receitas a pessoas, agora vão ser vinculadas a usuários:

O models de receitas vai ficar assim:

from django.db import models  
from datetime import datetime  
from django.contrib.auth.models import User  
  
  
class Receita(models.Model):  
  
 pessoa = models.ForeignKey(User, on\_delete=models.CASCADE)  
  
 nome\_receita = models.CharField(max\_length=200)  
 ingredientes = models.TextField()  
 modo\_preparo = models.TextField()  
 tempo\_preparo = models.IntegerField()  
 rendimento = models.CharField(max\_length=100)  
 categoria = models.CharField(max\_length=100)  
 date\_receita = models.DateTimeField(default=datetime.now, blank=True)  
 foto\_receita = models.ImageField(upload\_to='fotos/%d/%m/%Y', blank=True)  
 publicada = models.BooleanField(default=False)  
  
 def \_\_str\_\_(self):  
 return self.nome\_receita

Essa modificação tem que propagar para o banco de dados:

python manage.py makemigrations

python manage.py migrate

# Mensagens de sucesso e erro

Primeiramente vamos ao arquivo de settings.py incluir o código conforme orientação da documentação “messages”, no caso usaremos mensagem de erro e sucesso.

# messages  
from django.contrib.messages import constants as messages  
MESSAGE\_TAGS = {  
 messages.ERROR: 'danger',  
 messages.SUCCESS: 'success',  
}

Depois vamos criar um template, na parte de partials para as mensagens baseado nas messages do bootstrap:

{% if messages %}  
 {% for message in messages %}  
 <div class="alert alert-{{ message.tags }}" role="alert">  
 {{ message }}  
 </div>  
 {% endfor %}  
{% endif %}

Depois de criada a partial, incluir nos arquivos html de login e cadastro:

<h2>Login</h2>  
 </div>  
 </div>  
</div>  
  
{% include 'partials/\_alertas.html' %}  
  
<div class="row">  
 <div class="col-12">  
 <div class="contact-form-area">  
 <form action="{% url 'login' %}" method="POST">

Por ultimo criar as mensagens no arquivo views de usuários:

from django.contrib import messages

def cadastro(request):  
 if request.method == 'POST':  
 nome = request.POST['nome']  
 email = request.POST['email']  
 senha = request.POST['password']  
 senha2 = request.POST['password2']  
 if not nome.strip():  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cadastro')  
 if not email.strip():  
 messages.error(request, 'O campo email não pode ficar em branco')  
 return redirect('cadastro')  
 if senha != senha2:  
 messages.error(request, 'As senhas não são iguais')  
 return redirect('cadastro')  
 if User.objects.filter(email=email).exists():  
 messages.error(request, 'Usuário já cadastrado')  
 return redirect('cadastro')  
 user = User.objects.create\_user(username=nome, email=email, password=senha)  
 user.save()  
 messages.success(request, 'Usuário cadastrado com sucesso')  
 return redirect('login')  
 else:  
 return render(request, 'usuarios/cadastro.html')

# Sugestão de refatoração

Onde há o código repetido usar uma função:

def campo\_vazio(campo):  
 return not campo.strip()

Criação de uma avaliação de nome de usuário repetido:

if User.objects.filter(username=nome).exists():  
 messages.error(request, 'O nome de usuário já existe')  
 return redirect('cadastro')

Foram criados mais testes de integridade para a criação das receitas

def cria\_receita(request):  
 if request.method == 'POST':  
 nome\_receita = request.POST['nome\_receita']  
 if campo\_vazio(nome\_receita):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 ingredientes = request.POST['ingredientes']  
 if campo\_vazio(ingredientes):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 modo\_preparo = request.POST['modo\_preparo']  
 if campo\_vazio(modo\_preparo):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 tempo\_preparo = request.POST['tempo\_preparo']  
 if campo\_vazio(tempo\_preparo):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 rendimento = request.POST['rendimento']  
 if campo\_vazio(rendimento):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 categoria = request.POST['categoria']  
 if campo\_vazio(categoria):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 foto\_receita = request.FILES['foto\_receita']  
 user = get\_object\_or\_404(User, pk=request.user.id)  
 # maneira de pegar o usuário na requisição  
 receita = Receita.objects.create(pessoa=user, nome\_receita=nome\_receita,  
 ingredientes=ingredientes, modo\_preparo=modo\_preparo,  
 tempo\_preparo=tempo\_preparo, rendimento=rendimento,  
 categoria=categoria, foto\_receita=foto\_receita)  
 receita.save()  
 return redirect('dashboard')  
 else:  
 return render(request, 'usuarios/cria\_receita.html')

# Deletando receitas

Como deletar um item criado por um usuário:

1. Criar um botão para deletar no template de dashboard:

<h5>{{ receita.nome\_receita }}</h5>  
<a href="{% url 'deleta\_receita' receita.id %}" type="button" class="btn btn-danger">  
 Deletar  
</a>

1. Criar uma url para deletar:

path('deleta/<int:receita\_id>', views.deleta\_receita, name='deleta\_receita'),

1. Criar um view:

def deleta\_receita(request, receita\_id):  
 receita = get\_object\_or\_404(Receita, pk=receita\_id)  
 receita.delete()  
 return redirect('dashboard')

# deletando um APP

Para deletar um app não basta tirar do settings, tem que retirar também os vínculos com as migrações (no caso comparar a migração 2 com a 5)

# Editar dados de uma receita

Para habilitar a edição de uma receita:

1. Criar um botão para editar no template de dashboard:

<a href="{% url 'edita\_receita' receita.id %}" type="button" class="btn btn-info">  
 Editar  
</a>

1. Criar uma url para editar:

path('edita/<int:receita\_id>', views.edita\_receita, name='edita\_receita'),

1. Criar um view:

def edita\_receita(request, receita\_id):  
 receita = get\_object\_or\_404(Receita, pk=receita\_id)  
  
 receita\_a\_editar = {  
 'receita': receita  
 }  
 return render(request, 'usuarios/edita\_receita.html', receita\_a\_editar)

1. Cria um formulário html para editar:

O item principal do formulário é a ação e o método:

A ação será com uma url e o método é POST

Código omitido

<form action="{% url 'alualiza\_receita' %}" method="post" enctype="multipart/form-data">  
 {% csrf\_token %}  
 <input type="text" id="receita\_id" class="form-control" name="receita\_id" value="{{receita.id}}" hidden>  
 <div class="row">  
 <div class="col-12 col-lg-12">  
 <label for="nome\_receita"><b>Título da receita</b></label>

Código omitido

O que vai acarretar na criação de um novo path no arquivo de url e consequentemente uma view:

path('atualiza\_receita', views.atualiza\_receita, name='atualiza\_receita'),

def atualiza\_receita(request):  
 if request.method == 'POST':  
 receita\_id = request.POST['receita\_id']  
 r = Receita.objects.get(pk=receita\_id)  
 r.nome\_receita = request.POST['nome\_receita']  
 r.ingredientes = request.POST['ingredientes']  
 r.modo\_preparo = request.POST['modo\_preparo']  
 r.tempo\_preparo = request.POST['tempo\_preparo']  
 r.rendimento = request.POST['rendimento']  
 r.categoria = request.POST['categoria']  
 if 'foto\_receita' in request.FILES:  
 r.foto\_receita = request.FILES['foto\_receita']  
 r.save()  
 return redirect('dashboard')

# melhoria de arquitetura

## Templates

Os arquivos html que são referentes às receitas vão para uma pasta receitas, assim como os arquivos html de usuários e partial estão. Ao mudar os arquivos para a pasta será necessária a atualização do arquivo views. (a refatoração do pycharm fez automaticamente)

## Views

Algumas funções relativas à manipulação de receitas estão no arquivo de usuários, deve ser mudado, mas sendo necessário atualizar o path.

Um procedimento de organização possível é de separar todas as funções de views em uma pasta com arquivos temáticos e criar um arquivo de inicialização \_\_init\_\_.py importando os métodos dos arquivos, quando atualizar as paths só precisa tirar as referências de views.

Busca.py

from receitas.models import Receita  
from django.shortcuts import render  
  
  
def busca(request):  
 lista\_receitas = Receita.objects.order\_by('-date\_receita').filter(publicada=True)  
  
 if 'search' in request.GET: # o nome está assim por causa do botão do html,  
 # poderia estar em português  
 nome\_a\_buscar = request.GET['search']  
 lista\_receitas = lista\_receitas.filter(nome\_receita\_\_icontains=nome\_a\_buscar)  
  
 dados = {  
 'receitas': lista\_receitas,  
 }  
  
 return render(request, 'receitas/buscar.html', dados)

urls.py

from django.urls import path  
from .views import \*  
  
urlpatterns = [  
 path('', index, name='index'),  
 path('<int:receita\_id>', receita, name='receita'),  
 path('busca', busca, name='buscar'),  
 path('cria/receita', cria\_receita, name='cria\_receita'),  
 path('deleta/<int:receita\_id>', deleta\_receita, name='deleta\_receita'),  
 path('edita/<int:receita\_id>', edita\_receita, name='edita\_receita'),  
 path('atualiza\_receita', atualiza\_receita, name='atualiza\_receita'),  
]

\_\_init\_\_.py

from .receitas import \*  
from .busca import \*

receitas.py

from django.shortcuts import render, get\_object\_or\_404, redirect  
from receitas.models import Receita  
from django.contrib.auth.models import User  
from django.contrib import messages  
  
  
def index(request):  
 receitas = Receita.objects.order\_by('-date\_receita').filter(publicada=True)  
  
 dados = {  
 'receitas' : receitas  
 }  
  
 return render(request, 'receitas/index.html', dados)  
  
  
def receita(request, receita\_id):  
 receita = get\_object\_or\_404(Receita, pk=receita\_id)  
  
 receita\_a\_exibir = {  
 'receita': receita  
 }  
  
 return render(request, 'receitas/receita.html', receita\_a\_exibir)  
  
  
def cria\_receita(request):  
 if request.method == 'POST':  
 nome\_receita = request.POST['nome\_receita']  
 if campo\_vazio(nome\_receita):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 ingredientes = request.POST['ingredientes']  
 if campo\_vazio(ingredientes):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 modo\_preparo = request.POST['modo\_preparo']  
 if campo\_vazio(modo\_preparo):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 tempo\_preparo = request.POST['tempo\_preparo']  
 if campo\_vazio(tempo\_preparo):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 rendimento = request.POST['rendimento']  
 if campo\_vazio(rendimento):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 categoria = request.POST['categoria']  
 if campo\_vazio(categoria):  
 messages.error(request, 'O campo nome não pode ficar em branco')  
 return redirect('cria\_receita')  
  
 foto\_receita = request.FILES['foto\_receita']  
 user = get\_object\_or\_404(User, pk=request.user.id)  
 # maneira de pegar o usuário na requisição  
 receita = Receita.objects.create(pessoa=user, nome\_receita=nome\_receita,  
 ingredientes=ingredientes, modo\_preparo=modo\_preparo,  
 tempo\_preparo=tempo\_preparo, rendimento=rendimento,  
 categoria=categoria, foto\_receita=foto\_receita)  
 receita.save()  
 return redirect('dashboard')  
 else:  
 return render(request, 'receitas/cria\_receita.html')  
  
  
def deleta\_receita(request, receita\_id):  
 receita = get\_object\_or\_404(Receita, pk=receita\_id)  
 receita.delete()  
 return redirect('dashboard')  
  
  
def edita\_receita(request, receita\_id):  
 receita = get\_object\_or\_404(Receita, pk=receita\_id)  
  
 receita\_a\_editar = {  
 'receita': receita  
 }  
 return render(request, 'receitas/edita\_receita.html', receita\_a\_editar)  
  
  
def atualiza\_receita(request):  
 if request.method == 'POST':  
 receita\_id = request.POST['receita\_id']  
 r = Receita.objects.get(pk=receita\_id)  
 r.nome\_receita = request.POST['nome\_receita']  
 r.ingredientes = request.POST['ingredientes']  
 r.modo\_preparo = request.POST['modo\_preparo']  
 r.tempo\_preparo = request.POST['tempo\_preparo']  
 r.rendimento = request.POST['rendimento']  
 r.categoria = request.POST['categoria']  
 if 'foto\_receita' in request.FILES:  
 r.foto\_receita = request.FILES['foto\_receita']  
 r.save()  
 return redirect('dashboard')  
  
  
def campo\_vazio(campo):  
 return not campo.strip()

# Paginação

A paginação será feita alterando o código da view de index no arquivo receita.py

Usando o django.core.paginator

def index(request):  
 receitas = Receita.objects.order\_by('-date\_receita').filter(publicada=True)  
  
 paginator = Paginator(receitas, 3)  
 page = request.GET.get('page')  
 receitas\_por\_pagina = paginator.get\_page(page)  
  
 dados = {  
 'receitas': receitas\_por\_pagina  
 }

vamos inserir a paginação no html do index

<!-- ##### Pagination ##### -->  
<section class="top-catagory-area section-padding-20-0">  
 <div class="container">  
  
 {% if receitas.has\_other\_pages %}  
 <ul class="pagination">  
  
 {% if receitas.has\_previous %}  
 <li class="page-item">  
 <a href="?page={{ receitas.previous\_page\_number }}" class="page-link">&laquo;</a>  
 </li>  
 {% else %}  
 <li class="page-item disabled">  
 <a class="page-link">&laquo;</a>  
 </li>  
 {% endif %}  
  
 {% for pagina in receitas.paginator.page\_range %}  
 {% if receitas.number == pagina %}  
 <li class="page-item active">  
 <a class="page-link">{{ pagina }}</a>  
 </li>  
 {% else %}  
 <li class="page-item">  
 <a href="?page={{ pagina }}" class="page-link">{{ pagina }}</a>  
 </li>  
 {% endif %}  
 {% endfor %}  
  
 {% if receitas.has\_next %}  
 <li class="page-item">  
 <a href="?page={{ receitas.next\_page\_number }}" class="page-link">&raquo;</a>  
 </li>  
 {% else %}  
 <li class="page-item disabled">  
 <a class="page-link">&raquo;</a>  
 </li>  
 {% endif %}  
 </ul>  
 {% endif %}  
 </div>  
</section>  
<!-- ##### Pagination End ##### -->

# Verificando superuser

Uma forma de verificar se o usuário logado é superuser e caso seja, um link é exibido para acessar a view de admin:

Para isso basta inserir no partial do menu uma regra

<ul>

{% if user.is\_authenticated %}

<li><a href="{% url 'index' %}">Página principal</a></li>

<li><a href="{% url 'dashboard' %}">Minhas receitas</a></li>

<li><a href="{% url 'cria\_receita' %}">Criar receita</a></li>

{% if user.is\_superuser %}

<li><a href="{% url 'admin:index' %}">Admin</a></li>

{% endif %}

<li><a href="{% url 'logout' %}">Logout</a></li>

{% else %}

<li><a href="{% url 'cadastro' %}">Cadastro</a></li>

<li><a href="{% url 'login' %}">Login</a></li>

{% endif %}

</ul>

Desta forma, não precisamos digitar na url /admin para ter acesso a área do Admin, caso usuário logado seja uma superuser.

# Pasta apps e docstring

Para melhorar mais um pouco a arquitetura da nossa aplicação, vamos colocar os apps todos na página apps, mas para isso temos que inserir código no arquivo settings:

import sys

PROJECT\_ROOT = os.path.dirname(\_\_file\_\_)  
sys.path.insert(0, os.path.join(PROJECT\_ROOT, '/../apps'))

# Segundo projeto

Nesse Segundo projeto é imitando um site de venda de passagens aéreas para simular o controle de dados e preenchimento de um formulário.

Após o procedimento de criação de um projeto em django, vamos criar um app chamado passagens:

> python manage.py startapp passagens

Após isso, o cadastro dele no arquivo de settings e direcionamento no path para o site

## Django Form

Inicialmente, no app de passagens criamos o arquivo forms.py. com o tempo vamos desenvolver mais a classe criada:

from django import forms  
  
  
class PassagemForms(forms.Form):  
 origem = forms.CharField(label='Origem', max\_length=100)  
 destino = forms.CharField(label='Destino', max\_length=100)

Em seguida, vamos importar no views

from django.shortcuts import render  
from passagens.forms import PassagemForms  
  
  
def index(request):  
 form = PassagemForms()  
 contexto = {'form': form}  
 return render(request, 'index.html', context=contexto)

Depois colocamos no template html com método POST

<body>  
 <section class="container mt-2">  
 <h1>Passagem</h1>  
  
 <form action="{% url 'minha\_consulta' %}" method="post">  
 {% csrf\_token %}  
 {{ form }}  
 <input type="button" value="OK" class="btn btn-success">  
 </form>  
  
 </section>  
</body>

Para poder pegar as informações do formulário para checar, criamos uma nova url (path), uma nova view e criar o html para a exibição (minha consulta):

from django.urls import path  
from . import views  
  
urlpatterns = [  
 path('', views.index, name='index'),  
 path('minha\_consulta', views.revisao\_consulta, name='minha\_consulta'),  
]

def revisao\_consulta(request):  
 if request.method == 'POST':  
 form = PassagemForms(request.POST)  
 contexto = {'form': form}  
 return render(request, 'minha\_consulta.html', context=contexto)

{% extends 'base.html' %}  
  
{% block content %}  
  
 <p>Sua Origem é: {{ form.origem.value }}</p>  
 <p>Seu Destino é: {{ form.destino.value }}</p>  
 <p>Sua data de ida é: {{ form.data\_ida.value }}</p>  
 <p>Sua data de volta é: {{ form.data\_volta.value }}</p>  
  
{% endblock %}

## Django tempus dominus

Forma para controlar a entrada de dados de tempo (por enquanto está funcionando no django 3) provavelmente teremos que usar o bootstrap para não usar esse módulo.

1. instalar: pip install django-tempus-dominus
2. colocar o tempus\_dominus nos apps instalados no arquivo de settings

INSTALLED\_APPS = [  
 'tempus\_dominus',  
 'passagens',

1. no final do settings colocar a configuração do tempus para ficar em português

TEMPUS\_DOMINUS\_LOCALIZE = True

1. Colocar no arquivo forms a parte de data e importar o tempus\_dominus

from django import forms  
from tempus\_dominus.widgets import DatePicker  
  
  
class PassagemForms(forms.Form):  
 origem = forms.CharField(label='Origem', max\_length=100)  
 destino = forms.CharField(label='Destino', max\_length=100)  
 data\_ida = forms.DateField(label='Ida', widget=DatePicker())  
 data\_volta = forms.DateField(label='Volta', widget=DatePicker())

1. Colocar código específico na base do html, importar bootstrap e jquery

{# Include Bootstrap 4 and jQuery #}

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

{# Django Tempus Dominus assets are included in `{{ form.media }}` #}

{{ form.media }}

## Novos campos no formulário

Inserindo novas possibilidades de campos no formulário:

from django import forms  
from tempus\_dominus.widgets import DatePicker  
from datetime import datetime  
from .classe\_viagem import tipos\_de\_classe  
  
  
class PassagemForms(forms.Form):  
 origem = forms.CharField(label='Origem', max\_length=100)  
 destino = forms.CharField(label='Destino', max\_length=100)  
 data\_ida = forms.DateField(label='Ida', widget=DatePicker())  
 data\_volta = forms.DateField(label='Volta', widget=DatePicker())  
 data\_pesquisa = forms.DateField(label='Data Pesquisa', disabled=True, initial=datetime.today())  
 classe\_viagem = forms.ChoiceField(label='Classe do Vôo', choices=tipos\_de\_classe)  
 informacoes = forms.CharField(  
 label='Informações extras',  
 max\_length=300,  
 widget=forms.Textarea(),  
 required=False,  
 )  
 email = forms.EmailField(label='E-mail', max\_length=150)

Temos que modificar o código do index:

<form action="{% url 'minha\_consulta' %}" method="post">  
 {% csrf\_token %}  
 {% for field in form.visible\_fields %}  
 <div>  
 <label for="{{ field.id\_for\_label }}">{{ field.label }}</label>  
 {{ field }}  
 </div>  
 {% endfor %}  
 <input type="submit" value="OK" class="btn btn-success">  
</form>

Agora para poder organizar a aparência do formulário utilizaremos o django-widget-tweaks que deve ser instalado:

pip install django-widget-tweaks

Depois, conforme documentação, cadastramos como app instalado:

INSTALLED\_APPS = [

...

'widget\_tweaks',

...

]

No html inserimos no início:

{% load widget\_tweaks %}

E no formulário Podemos cadastrar a classe do bootstrap:

{{ field|add\_class:'form-control' }}

## Validação dos dados do formulário

A validação se dá por meio de função de validação no mesmo arquivo do formulário, necessariamente começando com o nome “clean\_”:

def clean\_origem(self):  
 origem = self.cleaned\_data.get('origem')  
 if any(char.isdigit() for char in origem):  
 raise forms.ValidationError('Campo Origem inválido! \nNão pode usar número')  
 else:  
 return origem

No caso de ter muitas checagens iguais nos campos, criaremos um método genérico e as checagens mais duras serão feitas em um arquivo validation.

No arquivo forms

from .validation import \*

(...)

def clean(self):  
 origem = self.cleaned\_data.get('origem')  
 destino = self.cleaned\_data.get('destino')  
 data\_ida = self.cleaned\_data.get('data\_ida')  
 data\_volta = self.cleaned\_data.get('data\_volta')  
 data\_pesquisa = self.cleaned\_data.get('data\_pesquisa')  
 lista\_erros = {}  
  
 campo\_com\_numero(origem, 'origem', lista\_erros)  
 campo\_com\_numero(destino, 'destino', lista\_erros)  
 origem\_destino\_iguais(origem, destino, lista\_erros)  
 data\_ida\_maior\_que\_data\_volta(data\_ida, data\_volta, lista\_erros)  
 data\_ida\_menor\_que\_data\_hoje(data\_ida, data\_pesquisa, lista\_erros)  
  
 if lista\_erros is not None:  
 for erro in lista\_erros:  
 mensagem\_erro = lista\_erros[erro]  
 self.add\_error(erro, mensagem\_erro)  
  
 return self.cleaned\_data

No arquivo validation:

def origem\_destino\_iguais(origem, destino, lista\_erros):  
 *"""Verifica se origem e destino são iguais"""* if origem == destino:  
 lista\_erros['destino'] = "Origem e destino não podem ser iguais"  
  
  
def campo\_com\_numero(valor\_do\_campo, nome\_do\_campo, lista\_erros):  
 *"""Verifica se tem caracter numérico no campos"""* if any(char.isdigit() for char in valor\_do\_campo):  
 lista\_erros[nome\_do\_campo] = 'Campo não pode ter números'  
  
  
def data\_ida\_maior\_que\_data\_volta(data\_ida, data\_volta, lista\_erros):  
 *"""Verificação das datas lançadas para que a ida não seja maior que a data de volta"""* if data\_ida > data\_volta:  
 lista\_erros['data\_volta'] = "Data de ida não pode ser maior que a data de volta!"  
  
  
def data\_ida\_menor\_que\_data\_hoje(data\_ida, data\_pesquisa, lista\_erros):  
 *"""Verificação da data de ida não seja menor que a data da pesquisa"""* if data\_ida > data\_pesquisa:  
 lista\_erros['data\_volta'] = "Data de ida não pode ser menor que a data de hoje!"

depois é necessário incluir na view a validação:

def revisao\_consulta(request):  
 if request.method == 'POST':  
 form = PassagemForms(request.POST)  
 if form.is\_valid():  
 contexto = {'form': form}  
 return render(request, 'minha\_consulta.html', context=contexto)  
 else:  
 contexto = {'form': form}  
 return render(request, 'index.html', context=contexto)

No html vamos colocar um método para aparecer a mensagem de erro:

{% for error in field.errors %}  
 <section class="alert alert-danger" role="alert">  
 {{ field.errors }}  
 </section>  
{% endfor %}

# Criando um formulário através de um modelo

Foi disponibilizada uma pasta models para adiantar o conteúdo.

Os modelos de classes que serão as tabelas foram cedidas, em 3 arquivos distintos e um arquivo \_\_init\_\_ que une as 3.

Depois de feita a migração e criado o banco de dados, vamos ao forms, onde será diferente do método anterior.

from .models import Passagem

(...)

class PassagemForms(forms.ModelForm):  
 data\_pesquisa = forms.DateField(label='Data Pesquisa', disabled=True, initial=datetime.today())  
  
 class Meta:  
 model = Passagem  
 fields = '\_\_all\_\_'  
 labels = {'data\_ida': 'Data de ida', 'data\_volta': 'Data de volta',  
 'informacoes': 'Informações', 'classe\_viagem': 'Classe do vôo'}  
 widgets = {  
 'data\_ida': DatePicker(),  
 'data\_volta': DatePicker(),  
 }  
  
 def clean(self):  
 origem = self.cleaned\_data.get('origem')  
 destino = self.cleaned\_data.get('destino')  
 data\_ida = self.cleaned\_data.get('data\_ida')  
 data\_volta = self.cleaned\_data.get('data\_volta')  
 data\_pesquisa = self.cleaned\_data.get('data\_pesquisa')  
 lista\_erros = {}  
  
 campo\_com\_numero(origem, 'origem', lista\_erros)  
 campo\_com\_numero(destino, 'destino', lista\_erros)  
 origem\_destino\_iguais(origem, destino, lista\_erros)  
 data\_ida\_maior\_que\_data\_volta(data\_ida, data\_volta, lista\_erros)  
 data\_ida\_menor\_que\_data\_hoje(data\_ida, data\_pesquisa, lista\_erros)  
  
 if lista\_erros is not None:  
 for erro in lista\_erros:  
 mensagem\_erro = lista\_erros[erro]  
 self.add\_error(erro, mensagem\_erro)  
  
 return self.cleaned\_data

Lembrando que ao criar os formulários no forms eles devem ser importados no views.