

PYTHON E DJANGO FRAMEWORK

Raíssa Azevedo

PARA
Beginners BR

APRESENTADO POR
Raíssa Azevedo

APRESENTAÇÃO

Neste pequeno ebook você vai encontrar todos os passos básicos para desenvolver um sistema web utilizando Django Framework.

Os passos vão desde criação de modelos relacionais, banco de dados, configuração do ambiente, instalação, aplicação de imagens, charts, PDF até o deploy de uma aplicação Django.

Não se esqueça de parar na parte de “Comandos importantes” que sempre vai precisar.

[Raíssa Azevedo](#)

COMANDOS IMPORTANTES

- Instalar o django

```
pip install django ou pip3 install django
```

- Para criar um projeto django

```
django-admin startproject nomedoprojet .
```

- Para criar uma aplicação django:

```
django-admin startapp nameapp
```

- Para rodar uma aplicação django no servidor

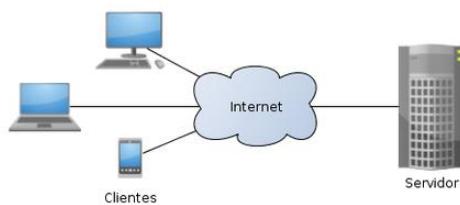
```
python3 manage.py runserver
```

CONCEITOS BÁSICOS

Eis alguns conceitos básicos que é importante entender antes de iniciar a aplicação django:

- **Arquitetura Cliente-Servidor**

É um modelo de computação distribuída em que um sistema é dividido em duas partes principais: o cliente, que é responsável pela interação com o usuário e a interface gráfica, e o servidor, que é responsável pelo processamento dos dados e pela lógica de negócios.



O cliente envia solicitações ao servidor, que processa as informações solicitadas e envia as respostas de volta ao cliente. Essa arquitetura é amplamente utilizada em redes de computadores e na internet, permitindo que aplicativos e serviços sejam executados de forma escalável, confiável e segura.

- **Protocolo HTTP**

HTTP (*Hypertext Transfer Protocol*) é um protocolo de comunicação usado para transferir dados pela internet. Ele define como os dados serão formatados e transmitidos entre clientes e servidores web.

Os verbos HTTP são métodos que definem a ação a ser realizada em um determinado recurso. Os principais verbos HTTP são:

GET: solicita um recurso ao servidor;
POST: envia dados ao servidor para serem processados;
PUT: atualiza um recurso existente no servidor;
DELETE: remove um recurso do servidor;
HEAD: solicita apenas informações sobre o recurso, sem obtê-lo;
OPTIONS: obtém informações sobre os métodos HTTP suportados pelo servidor.

Existem outros verbos HTTP menos comuns, como **PATCH**, **TRACE**, **CONNECT**, entre outros.

- **Backend x Frontend x Fullstack**

Backend, frontend e fullstack são termos usados para descrever diferentes áreas de desenvolvimento de software.

O **Backend** é a parte do software que lida com a lógica de negócios e o processamento de dados do aplicativo. Isso inclui o gerenciamento de bancos de dados, a lógica de autenticação, a comunicação com APIs externas, entre outras coisas.

O **Frontend** é a parte do software que lida com a interface do usuário. Isso inclui o design da interface, a implementação de interações de usuário, a integração com APIs externas, entre outras coisas.

Fullstack é um termo usado para descrever desenvolvedores que têm conhecimento tanto em frontend quanto em backend. Esses desenvolvedores são capazes de lidar com todo o processo de desenvolvimento de software, desde o design da interface do usuário até o gerenciamento do banco de dados e da lógica de negócios. Em outras palavras, eles são capazes de trabalhar com todas as camadas do aplicativo.

Em resumo, Backend lida com a lógica do servidor, Frontend lida com a interface do usuário e Fullstack é capaz de trabalhar com ambas as partes do software.

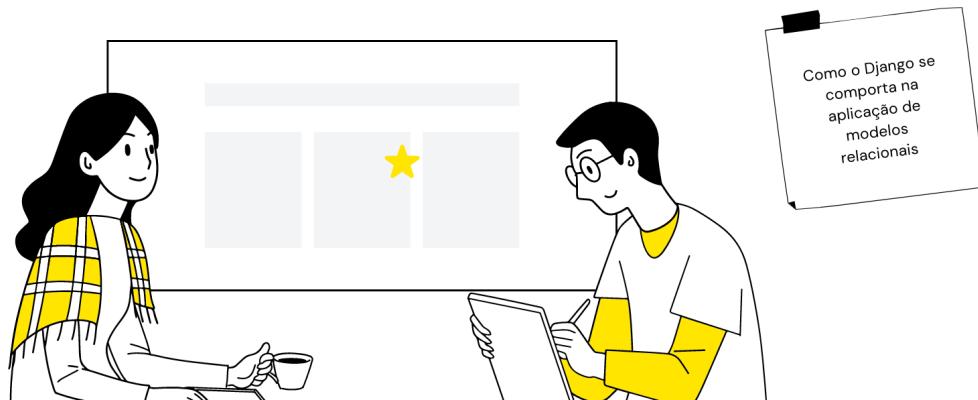
- **Programação estática x dinâmica**

Programação estática e dinâmica são abordagens diferentes para criar conteúdo web.

A **programação estática** envolve a criação manual de cada página web individualmente, usando HTML, CSS e outras linguagens de marcação. As páginas são criadas uma vez e permanecem inalteradas até que sejam atualizadas manualmente. Isso torna a programação estática rápida e segura, mas não permite muita interatividade.

A **programação dinâmica**, por outro lado, é criada usando linguagens de programação que permitem que o conteúdo seja gerado automaticamente a partir de um banco de dados ou de outras fontes de dados. Isso permite que os sites sejam mais interativos e personalizados para cada usuário, pois o conteúdo é gerado de forma dinâmica em tempo real. No entanto, a programação dinâmica pode ser mais lenta e vulnerável a ataques de segurança, dependendo da implementação.

Django e modelos relacionais



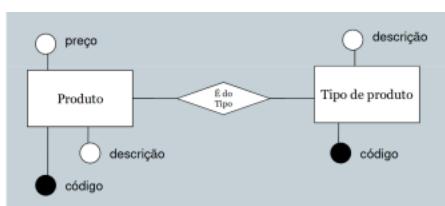
[Raíssa Azevedo](#)

Modelos Relacionais no Django Framework

Modelagem conceitual, lógica e Física:

Geralmente as entidades nunca estão sozinhas, é comum que estejam associadas umas com as outras, fornecendo uma descrição mais rica dos elementos do modelo. Um relacionamento pode acontecer entre uma, duas ou várias entidades.

Ex: Relacionamento entre entidade Produto e Tipo Produto:

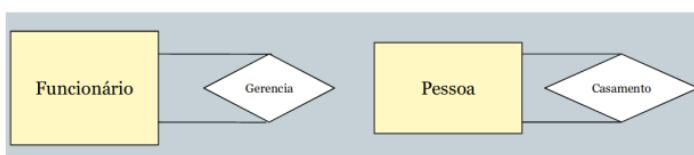


Outros conceitos:

Chave estrangeira – Também conhecido como foreign key ou fk. É um atributo presente em uma entidade que indica um relacionamento e representa a chave primária de uma entidade

Grau de relacionamento – Indica a quantidade de entidades ligadas a um relacionamento. Pode ser: Unário, binário ou Ternário.

Unário – Grau 1, é quando uma entidade se relaciona com ela mesma.

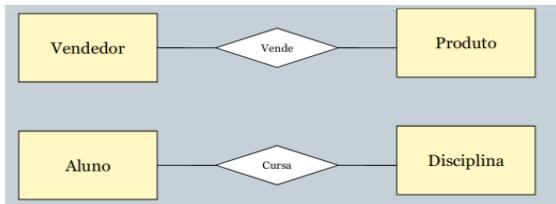


Exemplo onde um funcionário é gerenciado por outro funcionário.

Exemplo onde uma pessoa casa com outra pessoa.

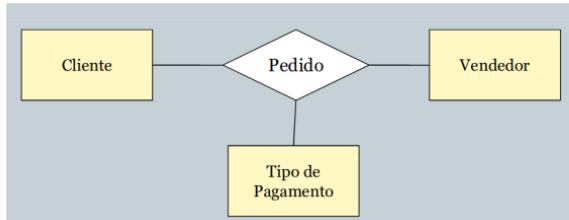
Binário – Grau 2, onde uma entidade se relaciona com outra entidade, é o tipo mais comum de relacionamento.

Ex:



Exemplo onde um vendedor vende produto e um aluno cursa disciplina.

Ternário – Grau 3, onde três entidades estão relacionadas por um mesmo relacionamento.



Exemplo onde um cliente fez um pedido que foi atendido por um vendedor e foi utilizado um tipo de pagamento para tal.

Obs: Dependendo da complexidade por haver mais graus.

PROJETO – DJANGO ORM

O Relacionamento um para um:

No relacionamento one to one (One2One). É um relacionamento pouco comum.

Após fazer as aplicações básicas no settings do projeto do django. Seguir para o models presente no app core.

```
from django.db import models

class Chassi(models.Model):
    numero = models.CharField('Chassi', max_length=16, help_text='Máximo:16 caracteres')

    class Meta:
        verbose_name = "Chassi"
        verbose_name_plural = 'Chassis'

    def __str__(self):
        return self.numero
```

O modelo Chassi implica em cadastrar o Chassi de veículos, que é a estrutura de identificação do veículo. O Chassi é único.

Em seguida, é necessário criar outra classe, onde ficará o relacionamento de fato.

```
class Carro(models.Model):
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')

    class Meta:
        verbose_name = 'Carro'
        verbose_name_plural = 'Carros'

    def __str__(self):
        return self.modelo
```

O relacionamento One to One significa que: Cada carro só pode relacionamento com 1 Chassi. Ou seja, um mesmo chassi também não pode estar relacionado a dois carros. Ou seja, um carro, um chassi (One2One).

No admin.py:

```
from django.contrib import admin
from .models import Chassi, Carro

@admin.register(Chassi)
class ChassiAdmin(admin.ModelAdmin):
    list_display = ['numero']

@admin.register(Carro)
class CarroAdmin(admin.ModelAdmin):
    list_display = ['modelo', 'chassi', 'descricao', 'preco']
```

Obs: Toda vez que criar um modelo executar as migrations.

```
python3 manage.py makemigrations  
python3 manage.py migrate
```

Em seguida executar os comandos para a criação de super usuário para administração do Banco de dados relacional:

```
python3 manage.py createsuperuser  
Usuário (leave blank to use 'raissa'):  
Endereço de email: rhaii.azevedo@gmail.com  
Password:  
Password (again):  
Superuser created successfully.
```

Em seguida, acessar a área administrativa através do servidor local do Django:

```
python3 manage.py runserver
```

Como não há um template criado ainda. Ficará somente a base administrativa padrão do Django:

The screenshot shows the Django Admin interface at the URL `127.0.0.1:8000/admin/`. The top navigation bar includes links for iGoogle, GitHub, Repl.it, and Quora. The main header reads "Administração do Django" and "BEM-VINDO(A), RAISSA". Below the header, there are two main sections: "AUTENTICAÇÃO E AUTORIZAÇÃO" containing "Grupos" and "Usuários" with "Adicionar" and "Modificar" buttons; and "CORE" containing "Carros" and "Chassis" with the same buttons. A sidebar on the left lists "Ações recentes" and "Minhas Ações", both currently showing "Nenhum disponível".

Agora todo o modelo está disponível para aplicação.

Ao cadastrar o Chassi, pode-se cadastrar o carro. Entretanto ao tentar cadastrar um novo carro com o mesmo Chassi o Django irá acusar que esse chassi já está cadastrado. Isso significa relacionamento One to One.

No django Shell:

```
python3 manage.py shell
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Corolla Cross XR 2.0>]>
>>> type(Carro)
<class 'django.db.models.base.ModelBase'>
>>>
```

É importante pedir o dir(Carros) para observar todos os atributos que podem ser utilizados e relacionados.

É possível fazer o o query:

```
>>> print(carros.query)
SELECT "core_carro"."id", "core_carro"."chassi_id", "core_carro"."modelo", "core_carro"."preco",
"core_carro"."descricao" FROM "core_carro"
>>>
```

Ao pesquisar o objeto carro:

```
>>> carro = Carro.objects.get(pk=1)
>>> carro
<Carro: Corolla Cross XR 2.0>
>>> type(carro)
<class 'core.models.Carro'>
```

É possível recuperar toda e qualquer informação fornecida através da primary key (pk) do carro através do Sheel.

```
>>> carro
<Carro: Corolla Cross XR 2.0>
>>> carro.modelo
'Corolla Cross XR 2.0'
>>> carro.preco
Decimal('139990.00')
>>> carro.descricao
'Sete airbags, câmera de ré com projeção na central multimídia (só as versões XRE, XRV e XRX têm guias dinâmicas), controle de estabilidade, controle de tração, assistente de partida em rampa, sensor de estacionamento traseiro, faróis com acendimento automático e ajuste de altura elétrico, faróis de neblina dianteiros em LED,'
>>> carro.chassi
<Chassi: 111sjdnsadn22535>
```

Mesmo a classe Chassi sendo construída separada da classe carro nos Models... Ela pôde ser acessada sem fazer o import no Shell devido ao relacionamento 1 para 1.

Chegando ao carro no Shell através do Chassi:

```
>>> from core.models import Chassi
>>> chassi = Chassi.objects.get(id=3)
>>> chassi
<Chassi: 111sjdnsadn22535>
>>> chassi.carro
<Carro: Corolla Cross XR 2.0>
>>> chassi.carro.modelo
'Corolla Cross XR 2.0'
```

Devido ao relacionamento One 2 One é possível acessar os dados tendo o número do Chassi fornecido na classe Chassi. Ou tendo a chave primária do carro fornecido em outra classe nos models.

O Relacionamento One to Many:

Imagine que todo carro possui uma montadora. Ou seja, existe a fabricante.

No models:

```
class Montadora(models.Model):
    nome = models.CharField('Nome', max_length=50)

    class Meta:
        verbose_name = "Nome"
        verbose_name_plural = 'Nomes'

    def __str__(self):
        return self.nome
```

Na class carro. Adicionar a montadora como uma chave estrangeira

```
class Carro(models.Model):
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')
    montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)

    class Meta:
        verbose_name = 'Carro'
        verbose_name_plural = 'Carros'

    def __str__(self):
        return f'{self.montadora} {self.modelo}'
```

Isso é feito dessa forma (**Foreign Key**). Porque cada carro possui apenas uma montadora. Mas uma montadora pode produzir diversos carros. Por isso ela deve ser apenas uma chave estrangeira dentro da classe de veículos.

Ao mexer nos models a migrations precisa ser criada novamente.

Se houver erros no models, e para corrigir precisa adicionar um default. E não quiser propor Blank ou default. O recomendado é apagar o banco de dados:

```
ls
core db.sqlite3 djangoorm manage.py venv
rm db.sqlite3
```

Além de apagar o banco de dados, apagar os arquivos **rm 000*** dentro da pasta migrations também.

Agora, executar as migrations novamente.

Após cadastrar os veículos através do servidor local do Django ou diretamente pelo shell

No Shell:

```
>>> from core.models import Carro  
>>> carros = Carro.objects.all()  
>>> carros  
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
```

Agora é possível pesquisar tudo relacionado ao carro.

```
>>> from core.models import Carro  
>>> carros = Carro.objects.all()  
>>> carros  
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>  
>>> fit = Carro.objects.get(pk=2)  
>>> fit  
<Carro: Honda Fit>  
>>> honda = fit.montadora  
>>> honda  
<Montadora: Honda>
```

Chegando no carro através da montadora:

```
>>> from core.models import Montadora  
>>> toyota = Montadora.objects.get(pk=1)  
>>> toyota  
<Montadora: Toyota>  
>>> carros = toyota.carro_set.all()  
>>> carros  
<QuerySet [<Carro: Toyota Corolla EX 2.0>]>
```

Como um montadora pode ter vários carros. Nas suas configurações ela vai ter o **carro_set** que consiste em uma lista com todos seus carros.

Isso significa a relação One to many.

O relacionamento muitos para muitos:

Existe um model carro. E cada carro pode ter vários motoristas. Como uma espécie de aluguel de carros. Portanto irá ser feita a vinculação do model carros com um model User.

Porque um carro pode ser dirigido por vários motoristas, e um motorista pode dirigir vários carros.

```
from django.contrib.auth import get_user_model  
  
class Carro(models.Model):  
    chassi = models.OneToOneField(Chassi, on_delete=models.CASCADE)  
    modelo = models.CharField('Modelo', max_length=30, help_text='Máximo:30 caracteres')  
    preco = models.DecimalField('Preço', max_digits=8, decimal_places=2)  
    descricao = models.TextField('Descrição', max_length=500, help_text='Máximo:500 caracteres')  
    montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)  
    motoristas = models.ManyToManyField(get_user_model())
```

Os usuários serão registrados na aba usuários da administração do Django:

The screenshot shows the Django Admin interface for the 'Users' model. At the top, there's a header bar with links for 'BEM-VINDO(A), RAISAA, VER O SITE / ALTERAR SENHA / ENCERRAR SESSÃO'. Below it, the breadcrumb navigation shows 'Início > Autenticação e Autorização > Usuários'. A search bar with placeholder 'Selecione usuário para modificar' and a 'Pesquisar' button is present. A 'ADICIONAR USUÁRIO +' button is also visible. The main area displays a table with columns: 'USUÁRIO', 'ENDEREÇO DE EMAIL', 'PRIMEIRO NOME', 'ÚLTIMO NOME', and 'MEMBRO DA EQUIPE'. The table contains three rows: 'Emilia' (Email: Emilia@gmail.com, First Name: Emilia, Last Name: Clarke, Member of Team: Red dot), 'Pedro' (Email: pedroca@gmail.com, First Name: Pedro, Last Name: Alcântara, Member of Team: Red dot), and 'raissa' (Email: rhail.azevedo@gmail.com, First Name: Raissa, Last Name: Azevedo, Member of Team: Green dot). A message at the bottom says '3 usuários'.

E a seleção para motoristas de cada carro fica dentro das especificações de cada carro:

The screenshot shows the Django Admin interface for the 'Carros' model, specifically for a 'Toyota Corolla EX 2.0'. The left sidebar has sections for 'AUTENTICAÇÃO E AUTORIZAÇÃO' (Grupos, Usuários) and 'CORE' (Carros, Chassis, Montadoras). The main panel is titled 'Modificar Carro' for 'Toyota Corolla EX 2.0'. It includes fields for 'Chassi' (with a dropdown menu showing '1111111111111111'), 'Modelo' (Corolla EX 2.0), 'Preço' (99880,50), 'Descrição' (Corolla 2.0, Cor: Preta, 4 portas, Completo, Ano: 2021), 'Montadora' (Toyota), and 'Motoristas' (Emilia, Pedro, raissa). A note at the bottom of the 'Motoristas' field says: 'Pressione "Control", ou "Command" no Mac, para selecionar mais de um.'

Assim é feito o controle de motoristas, os motoristas autorizado ficam com a coloração acinzentada no menu de descrição de cada veículo.

No Shell:

```
>>> from core.models import Carro
>>> carros = Carro.objects.all()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
>>> carro1 = carros.first()
>>> carro1
<Carro: Toyota Corolla EX 2.0>
>>> motors = carro1.motoristas.all()
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
```

Qual é, e quem são os motoristas do carro 2?

```
>>> carro2 = carros.last()
>>> carro2
<Carro: Honda Fit>
>>> motors2 = carro2.motoristas.all()
>>> motors2
<QuerySet [<User: Pedro>, <User: Emilia>]>
```

Isso é o relacionamento **many to many**, onde é possível ter vários carros com vários motoristas.

A medida que a complexidade de cada modelo relacional vai aumentando. Aumenta também a complexidade do query desse modelo.

```
>>> print(motors2.query)
SELECT "auth_user"."id", "auth_user"."password", "auth_user"."last_login", "auth_user"."is_superuser",
"auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email",
"auth_user"."is_staff", "auth_user"."is_active", "auth_user"."date_joined" FROM "auth_user" INNER JOIN
"core_carro_motoristas" ON ("auth_user"."id" = "core_carro_motoristas"."user_id") WHERE
"core_carro_motoristas"."carro_id" = 2
```

E se quiser saber a lista de carros que uma determinada lista de usuários dirige?

```
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
>>> m1 = motors.first()
>>> m1
<User: raissa>
>>> carros = Carro.objects.filter(motoristas=m1)
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>]>
>>> print(carros.query)
SELECT "core_carro"."id", "core_carro"."chassi_id", "core_carro"."modelo", "core_carro"."preco",
"core_carro"."descricao", "core_carro"."montadora_id" FROM "core_carro" INNER JOIN "core_carro_motoristas"
ON ("core_carro"."id" = "core_carro_motoristas"."carro_id") WHERE "core_carro_motoristas"."user_id" = 1
```

E se quiser saber qual a lista de carros dirigida pela lista inteira:

```
>>> motors
<QuerySet [<User: raissa>, <User: Emilia>]>
>>> carros = Carro.objects.filter(motoristas__in = motors)
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
```

Há uma repetição no nome dos carros, porque as vezes vários motoristas dirigem um mesmo carro.

Para filtrar:

```
>>> carros = Carro.objects.filter(motoristas__in = motors).distinct()
>>> carros
<QuerySet [<Carro: Toyota Corolla EX 2.0>, <Carro: Honda Fit>]>
>>>
```

Pronto agora o Shell exibe a lista sem repetição por causa do comando **distinct()** no final do comando de exibição dos carros dirigidos pela lista 1 de motoristas.

Aproveitando os recurso do Django models:

O que acontece com o Banco de Dados se deletar uma montadora?

The screenshot shows the Django Admin 'Administração do Django' header with the user 'BEM-VINDO(A), RAÍSSA' and links for 'VER O SITE / ALTERAR SENHA / ENCERRAR SESSÃO'. Below the header, the breadcrumb navigation shows 'Inicio > Core > Montadoras > Honda > Apagar'. A confirmation message asks 'Tem certeza?'. It lists the items to be deleted: 'Montadoras: 1', 'Carros: 1', and 'Relacionamentos carro-user: 2'. Under 'Objetos', it shows 'Montadora: Honda' with its details: 'Carro: Honda Fit', 'Relacionamento carro-user: Carro_motoristas object (3)', and 'Relacionamento carro-user: Carro_motoristas object (4)'. At the bottom are two buttons: a red 'Sim, eu tenho certeza' (Yes, I have certainty) and a grey 'Não, me leve de volta' (No, take me back).

Ou seja, o Django pede pela confirmação de deleção, e indica o que será perdido ao deletar uma das montadoras salvas no sistema. Que consiste em: 1 montadora, 1 carros e 2 relacionamentos de usuários que estão autorizados a dirigir.

Ou seja há uma remoção em cascata.

Isso ocorre porque:

```
montadora = models.ForeignKey(Montadora, on_delete=models.CASCADE)
```

Ao indicar a montadora no models.py. Colocamos a função **on_delete** = models.CASCADE. Isso orienta o django a fazer a remoção em cascada em tudo que tiver associado a montadora.

O **on_delete** em CASCADE deve ser adicionado tanto em Foreign Key quanto em many to many porque, se algo está vinculado a um objeto e esse objeto(montadora) deixa de existir, todo o vínculo também precisa ser removido.

Porque o carro depende da foreign key e da montadora como seus vínculos de identificação.

Ou seja, se você deleta o Chassi, o carro também precisa de deixar de existir dentro do sistema.



A importância dos testes automatizados do Django

Raíssa Azevedo

A IMPORTÂNCIA DE REALIZAR TESTES AUTOMATIZADOS

[Raíssa Azevedo](#)

Após mais um dia desvendando as funcionalidades do *Django Framework* me deparei com o arquivo **tests.py**. Estudando e me aprofundando a respeito cheguei aos seguintes passos de aprendizado:

Eu tinha um projeto de back-end realizado e publicado chamado [Fusion](#). Já tinha dado o projeto como finalizado. Mas, em todo o processo de criação, todos os testes que fiz buscando erros no código ou de lógica de programação foram manuais. Ou seja, eu precisava simular um servidor, adicionar os dados, e verificar o que me impedia de publicar.

Então, segue passos de realização de testes automatizados no modo de produção do Django Framework:

REALIZANDO TESTES DE USABILIDADE NO DJANGO

No Django cada aplicação criada irá conter um arquivo “tests.py”

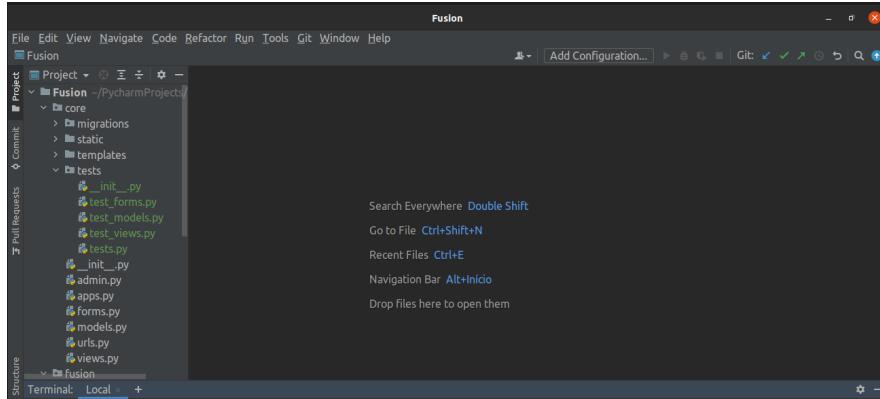
O recomendado é apagar esse arquivo e criar um diretório dentro da pasta de aplicação denominado “Tests”. Lá irá colocar todos os arquivos py que irá realizar os testes de usabilidade.

Nesse caso, a aplicação do meu projeto se chama core. E criei o diretório `tests` dentro dessa aplicação.

Obs: Todo arquivo de teste irá começar com **test_**

No arquivo `__init__.py`:

Foi realizado um teste do teste, que não está relacionado ao projeto, é somente para verificar se não há falha no Framework (99% de certeza que não haverá):



Instale as seguintes bibliotecas:

- `pip3 install model_mommy coverage`

```
from django.test import TestCase

# Teste de números

def add_num(num):
    return num + 1

class SimplesTestCase(TestCase):

    # Roda toda vez
    def setUp(self):
        self.numero = 41

    # Testa a unidade de código
    def test_add_num(self):
        valor = add_num(self.numero)
        self.assertTrue(valor == 42 )
```

Após escrever o seguinte código:

```
coverage run manage.py test
```

O teste unitário só irá retornar um valor **True** que indica que o teste automatizado está funcionando.

INICIANDO OS TESTES REAIS DO PROJETO

Na raiz do Projeto crie um arquivo **.coveragerc**

Esse arquivo irá indicar onde irá ocorrer os testes dentro do sistema criado

DENTRO DO ARQUIVO .COVERAGERC

```
[run]
source = .
```

Este “.” no código irá indicar que os testes irão ocorrer somente nas partes indicadas dentro do Projeto.

IGNORANDO ARQUIVOS QUE NÃO PRECISAM SER TESTADOS

O Framework Django por já testa boa parte do conteúdo presente no código. Sua responsabilidade é testar somente as criações novas do back-end como: Models (banco de dados), Formulários e Views.

Então, indique os arquivos a serem ignorados dessa forma:

```
[run]
source = .

omit =
  */__init__.py
  */settings.py
  */manage.py
  */wsgi.py
  */apps.py
  */urls.py
  */admin.py
  */migrations/*
  */tests/*
  */asgi.py
```

Todos esses arquivos e diretórios já são testados pelo Django, portanto não há necessidade de fazer um novo teste.

EXEMPLIFICANDO A EXECUÇÃO DE UM TESTE DE USABILIDADE

Testando o models.py:

Crie um arquivo no diretório de testes chamado **test_models.py**

Antes de tudo verifique tudo que precisa ser testado no seu models, executando:

```
coverage run manage.py test
```

Seguido de:

```
coverage html
```

Isso irá gerar um diretório chamado **Htmlcov**

Através do terminal abra esse diretório e execute o seguinte código:

```
python3 -m http.server
```

Isso irá gerar um servidor de relatório de testes que irá indicar tudo que precisa ser testado no seu models.

O servidor irá parecer mais ou menos como esse:

The screenshot shows a web browser window with the URL `0.0.0.0:8000`. The title bar says "Não seguro". Below the address bar, there are several bookmarks: iGoogle, GitHub - selat..., Repl.it - RaiiAz..., Início - Quora, Baixar, Como Pensar..., Ambiente Virt..., AprendaMais..., and another link. The main content area displays a "Coverage report: 100%" heading. Below it is a table showing coverage statistics for three modules:

Module	statements	missing	excluded	coverage
core/forms.py	15	0	0	100%
core/models.py	65	0	0	100%
core/views.py	23	0	0	100%
Total	103	0	0	100%

At the bottom of the table, it says `coverage.py v5.5, created at 2021-04-13 19:27-0300`.

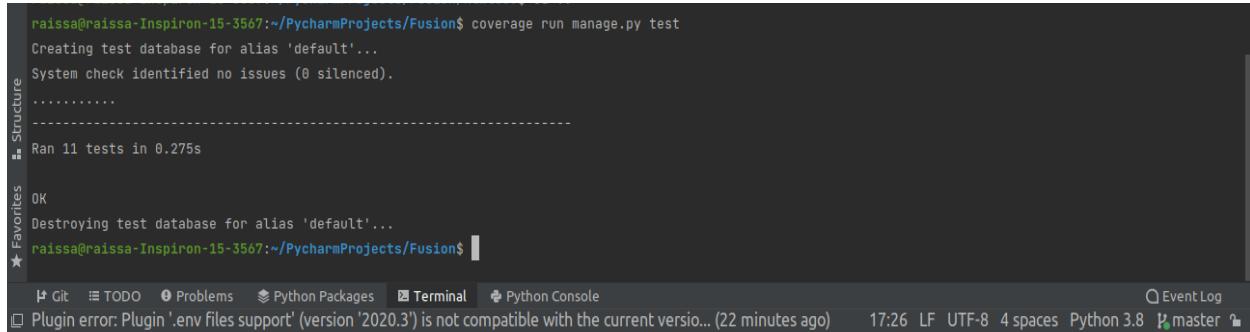
E irá indicar o que precisa ser testado, o que falta ser testado e a porcentagem de cobertura de testes. Tudo descrito em “Module” é um link que você pode abrir para encontrar mais dados específicos, como aqui no `models.py`:

The screenshot shows a coverage report for the file `core/models.py`. The title bar says "Coverage for core/models.py : 100%". Below it, there are status indicators: 65 statements, 65 run, 0 missing, and 0 excluded. The main content area shows the Python code with line numbers on the left. Lines 1 through 28 are shown, with each line having a green background, indicating that all statements have been executed. The code defines a `Base` class with `criados`, `modificado`, and `ativo` fields, and a `Meta` class with `abstract = True`. It also defines a `Servico` class with a `ICONES_CHOICES` list containing items like `('lni-cog', 'Engrenagem')`, `('lni-stats-up', 'Gráfico')`, etc.

Se houver algo que precisa ser testado, irá estar com coloração vermelha. E o que está verde, é tudo que está testado e sem presença de falhas. Esse relatório serve para indicar ao cliente a que pé anda seu projeto também.

COMO SABER SE HÁ ERROS NA EXECUÇÃO DE UM TESTE DE USABILIDADE

Ao executar um comando de testes, e se não houver falhas, o sistema irá te retornar uma resposta como esta:

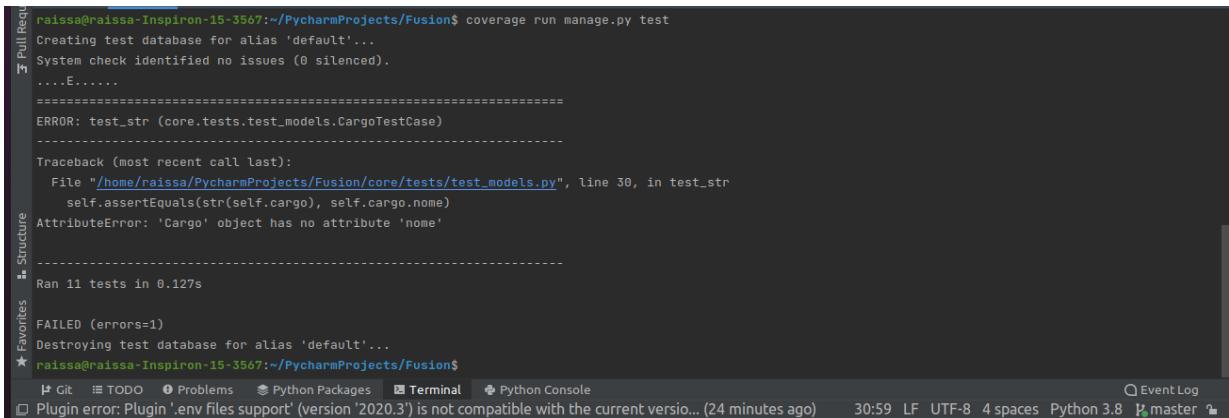


```
raissa@raissa-Inspiron-15-3567:~/PycharmProjects/Fusion$ coverage run manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 11 tests in 0.275s

OK
Destroying test database for alias 'default'...
raissa@raissa-Inspiron-15-3567:~/PycharmProjects/Fusion$
```

The screenshot shows the PyCharm interface with a terminal window open. The terminal output indicates a successful test run with 11 tests completed in 0.275 seconds, resulting in an 'OK' status. The PyCharm interface includes toolbars for Git, TODO, Problems, Python Packages, Terminal, and Python Console, along with an Event Log at the bottom.

Mas se houver erros, a resposta do sistema será assim:



```
raissa@raissa-Inspiron-15-3567:~/PycharmProjects/Fusion$ coverage run manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....E.....
=====
ERROR: test_str (core.tests.test_models.CargoTestCase)

-----
Traceback (most recent call last):
  File "/home/raissa/PycharmProjects/Fusion/core/tests/test_models.py", line 30, in test_str
    self.assertEquals(str(self.cargo), self.cargo.nome)
AttributeError: 'Cargo' object has no attribute 'nome'

-----
Ran 11 tests in 0.127s

FAILED (errors=1)
Destroying test database for alias 'default'...
raissa@raissa-Inspiron-15-3567:~/PycharmProjects/Fusion$
```

The screenshot shows the PyCharm interface with a terminal window open. The terminal output indicates a failed test run with 1 error (errors=1). A specific error is shown in the traceback: an AttributeError for the 'Cargo' object having no attribute 'nome'. The PyCharm interface includes toolbars for Git, TODO, Problems, Python Packages, Terminal, and Python Console, along with an Event Log at the bottom.

O sistema irá indicar a quantidade de falhas encontradas, qual arquivo está a falha e a linha exata onde está o erro.

Obs: Toda vez que for executar um novo comando de teste, lembre-se primeiro de apagar o diretório **htmlcov/** com o seguinte código:

```
rm -rf htmlcov
```

Para não haver conflito de dados de testes.

PORQUE EXECUTAR TESTES DE USABILIDADE É IMPORTANTE ?

Como citado anteriormente no texto, ao criar um projeto, você pode optar por testes manuais, mas só irá descobrir onde estão as falhas de programação no momento de inserir os dados e executar. O que torna o processo mais demorado. Uma vez o teste é executado, e indicando a presença de zero falhas, você poderá seguir para a próxima etapa do processo com a consciência tranquila de não haver erro no seu código. Além disso, um projeto com o comprovante de cobertura de testes 100% é um ponto positivo na carreira de qualquer programador.

Trabalhando com aplicações em tempo real

Raíssa Azevedo

Django Framework



[Raíssa Azevedo](#)

Como o Django se comporta em aplicações em Tempo Real ?

Entendendo as aplicações Realtime:

Uma aplicação realtime é a que há o compartilhamento de dados de uma ponta a outra a distância, e o conceito é: “O dado sai de um ponto estará disponível no outro ponto, no menor tempo possível”.

O PROJETO “realtime”:

Esse projeto vai contar com o banco de dados Redis.

```
pip3 install django channels django-bootstrap4 channels-redis  
pip3 freeze > requirements.txt
```

Porque não usar o Mysql ou o Postgre?

Porque precisa ser um sistema database que suporte atualização de dados em tempo real.

Em seguida, criar o projeto e a aplicação:

```
django-admin startproject realtime .  
django-admin startapp chat
```

Fazer todas as configurações necessárias no settings.py.

```
INSTALLED_APPS = [  
    'bootstrap4',  
    'channels',  
    'chat',
```

Por padrão o Django faz uso de uma aplicação WSGI.

Quando é necessário utilizar o realtime o WSGI não suporta. Portanto, é necessário criar uma aplicação ASGI:

```
# Configuração específica do Channels  
ASGI_APPLICATION = 'realtime.routing.application'
```

Além disso, é necessário configurar o Channels.

```
CHANNEL_LAYERS = {  
    'default': {  
        'BACKEND': 'channels_redis.core.RedisChannelLayer',  
        'CONFIG': {  
            'hosts': [('127.0.0.1', 6379)]  
        },  
    },  
}
```

6379 - É a porta do Redis.

Channels Layers – É preciso configurar o Channels para entender que é o Redis que vai comandar o fluxo de dados.

Configurando as Rotas e preparando o WebSocket:

Websocket é um interação entre o navegador e a sua aplicação para haver um comunicação realtime. Os navegadores atuais, possuem suporte ao Websocket.

Portanto, é possível utilizar esse recurso para criar um interação em tempo real com a aplicação.

Criar o arquivo **urls.py** na aplicação chat:

```
from django.urls import path  
  
from .views import IndexView, SalaView  
  
urlpatterns = [  
    path("", IndexView.as_view(), name='index'),  
    path('chat/<str:nome_sala>', SalaView.as_view(), name='sala'),  
]
```

Criar um arquivo **routing.py** na aplicação chat:

```
from django.urls import re_path  
  
from .consumers import ChatConsumer  
  
websocket_urlpatterns = [  
    re_path(r'^ws/chat/(?P<nome_sala>\w+)/$', ChatConsumer.as_asgi()),  
]
```

Essa é uma rota específica do Channels.

A rota é iniciada com **ws** que significa WebSocket. O ChatConsumer é quem irá fazer a ponte entre o navegador e a aplicação.

No arquivo de urls do Projeto:

```
from django.urls import path, include  
  
urlpatterns = [  
    path("", include('chat.urls')),  
    path('admin/', admin.site.urls),  
]
```

Criar um arquivo **routing.py** dentro da do diretório do Projeto realtime:

```
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack

from chat.routing import websocket_urlpatterns

application = ProtocolTypeRouter({
    'websocket': AuthMiddlewareStack(
        URLRouter(
            websocket_urlpatterns
        )
    ),
})
```

Trabalhando nas Views:

Na views do aplicação chat:

```
from django.views.generic import TemplateView
from django.utils.safestring import mark_safe
import json

class IndexView(TemplateView):
    template_name = 'index.html'

class SalaView(TemplateView):
    template_name = 'sala.html'

    def get_context_data(self, **kwargs):
        context = super(SalaView, self).get_context_data(**kwargs)
        context['nome_sala_json'] = mark_safe(
            json.dumps(self.kwargs['nome_sala']))
        return context
```

mark_safe – Do pacote SafeString é um função pra remover qualquer dado inseguro fornecido pelo usuário do chat.

Definindo os Templates:

No **Index.html**:

```
{% load bootstrap4 %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <link rel="shortcut icon" type="image/x-icon" href="/static/img/favicon.ico">  
    <title>Rai Chat</title>  
    {% bootstrap_css %}  
</head>  
<body>  
<div clas="container">  
    Qual sala de chat você gostaria de entrar?<br/>  
    <input id="nome_sala" name="nome_sala" type="text" size="100" placeholder="Nome da sala..."><br/>  
    {% buttons %}  
        <input id="botao" class="btn btn-primary" type="button" value="Entrar" />  
    {% endbuttons %}  
</div>  
<script>  
    document.querySelector('#nome_sala').focus();  
    document.querySelector('#nome_sala').onkeyup = function(e){  
        if(e.keyCode === 13){  
            document.querySelector('#botao').click();  
        }  
    };  
  
    document.querySelector('#botao').onclick = function(e){  
        var nome_sala = document.querySelector('#nome_sala').value;  
        if(nome_sala != ""){  
            window.location.pathname = '/chat/' + nome_sala + '/';  
        }else{  
            alert('Você precisa informar o nome da sala.');//  
            document.querySelector('#nome_sala').focus();  
        }  
    };  
</script>  
{% bootstrap_javascript jquery='full' %}  
</body>  
</html>
```

No **sala.html**:

```
{% load bootstrap4 %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <link rel="shortcut icon" type="image/x-icon" href="/static/img/favicon.ico">  
    <title>Rai Chat</title>  
    {% bootstrap_css %}  
</head>  
<body>  
    <div class="container">  
        <textarea id="sala" cols="70" rows="15"></textarea><br/>  
        <input id="texto" type="text" size="50"/><br/>  
        {% buttons %}  
        <input id="botao" type="button" value="Enviar" />  
        {% endbuttons %}  
    </div>  
  
{% bootstrap_javascript jquery='full' %}  
<script>  
    var nome_sala = {{ nome_sala_json }};  
  
    var chatSocket = new WebSocket(  
        'ws://' + window.location.host +  
        '/ws/chat/' + nome_sala + '/');  
  
    chatSocket.onmessage = function(e){  
        var dados = JSON.parse(e.data);  
        var mensagem = dados['mensagem'];  
        document.querySelector('#sala').value += (mensagem + '\n');  
    };  
  
    chatSocket.onclose = function(e){  
        console.error('O chat encerrou de forma inesperada.');//  
    };  
  
    document.querySelector('#texto').focus();  
    document.querySelector('#texto').onkeyup = function(e){  
        if(e.keyCode === 13){  
            document.querySelector('#botao').click();  
        }  
    };  
  
    document.querySelector('#botao').onclick = function(e){  
        var mensagemInput = document.querySelector('#texto');  
        var mensagem = mensagemInput.value;  
        chatSocket.send(JSON.stringify({  
            'mensagem': mensagem  
        }));  
        mensagemInput.value = " ";  
    };  
</script>  
</body>  
</html>
```

Criando o Consumer:

O elo de ligação entre a aplicação no navegador e o projeto realtime:

Criar o arquivo **consumers.py** dentro da aplicação chat:

```
from channels.generic.websocket import AsyncWebsocketConsumer
import json

class ChatConsumer(AsyncWebsocketConsumer):

    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['nome_sala']
        self.room_group_name = f'chat_{self.room_name}'

        # Entrar na sala
        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )

        await self.accept()

    async def disconnect(self, code):
        # Quando o usuário sai da sala
        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

    # Recebe mensagem do WebSocket
    async def receive(self, text_data=None, bytes_data=None):
        text_data_json = json.loads(text_data)
        mensagem = text_data_json['mensagem']

        # Envia a mensagem para a sala
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': mensagem
            }
        )

    # Recebe a mensagem da sala
    async def chat_message(self, event):
        mensagem = event['message']

        # Envia a mensagem para o WebSocket
        await self.send(text_data=json.dumps({
            'mensagem': mensagem
        }))


```

Quando trabalhamos com programação `async` é importante avisar que é assíncrona. Ela sendo assíncrona, todas as funções e comandos executados que dependem de algum retorno/finalização. É utilizado o `await`.

Rodando a aplicação em tempo real:

Iniciando o servidor Redis:

Vai até onde o servidor Redis está salvo dentro os diretórios:

```
redis-cli
```

Abra o terminal e digite:

```
pip3 manage.py migrate
```

Se houver erro ao executar o migrate:

```
pip uninstall django-channels
```

```
pip uninstall channels
```

```
pip install channels
```

Em seguida rodar a aplicação no servidor:

```
pip3 manage.py runserver
```



GERANDO ARQUIVO PDF COM DJANGO

Raíssa Azevedo

[Raíssa Azevedo](#)

Como gerar PDF na aplicação Django?

Existem várias formas de gerar documentos PDF com Django ou Python;

As bibliotecas necessárias para esse projeto são:

```
pip3 install django
django-admin startproject relatorio .
django-admin startapp core
```

Fazer as configurações necessárias no settings.py

Usar a biblioteca **Reportlab** recomendada nas documentações do Django:

```
pip3 install reportlab
python3 manage.py shell
import reportlab
```

Executar os seguintes passos para verificar se não houve nenhum erro na instalação da biblioteca.

Criar um arquivo de rotas na aplicação core:

```
from django.urls import path
from .views import IndexView

urlpatterns = [
    path('', IndexView.as_view(), name='index'),
]
```

Avisar a rotas do projeto:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('', include('core.urls')),
    path('admin/', admin.site.urls),
]
```

Criando os comandos para a geração do PDF:

Na views.py:

```
import io
from django.http import FileResponse
from django.views.generic import View

from reportlab.pdfgen import canvas

class IndexView(View):

    def get(self, request, *args, **kwargs):
        # Cria um arquivo para receber os dados e gerar o PDF
        buffer = io.BytesIO()

        # Cria o arquivo PDF
        pdf = canvas.Canvas(buffer)

        # Insere "coisas" no PDF
        pdf.drawString(100, 100, "Gerando PDF com Django")

        # Ao finalizar a inserção de coisas
        pdf.showPage()
        pdf.save()

        # Por fim, retornar o buffer para o inicio do arquivo
        buffer.seek(0)

    return FileResponse(buffer, as_attachment=True, filename='Relatorio1.pdf')

    # Abre o PDF direto no navegador
    return FileResponse(buffer, filename='Relatorio1.pdf')
```

Isso irá gerar um página em PDF.

O reportlab possui duas versões, a opensource com algumas limitações, e a versão paga.

Há outra forma de gerar PDF com Django:

Através da biblioteca **WeasyPrint** que é opensource e desenvolvido para usar com Python.

A partir de uma página HTML essa biblioteca printa a página e gera o arquivo PDF.

```
pip3 install WeasyPrint
```

No template de relatório.html:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Report - Django</title>
    <style>
        body {
            background: #0080ff;
        }
    </style>
</head>
<body>
    <h1>Report - Django</h1>
    {% for p in texto %}
        <p>{{ p }}</p>
    {% endfor %}
</body>
</html>
```

Na views.py:

```
# De uso para o WeasyPrint
from django.core.files.storage import FileSystemStorage
from django.template.loader import render_to_string
from django.http import HttpResponseRedirect
from weasyprint import HTML

class Index2View(View):
    def get(self, request, *args, **kwargs):
        texto = ['Geek University', 'Evoluua seu lado Geek', 'Programação Web com Python e Django']

        html_string = render_to_string('relatorio.html', {'texto': texto})

        html = HTML(string=html_string)
        html.write_pdf(target='/tmp/relatorio2.pdf')

        fs = FileSystemStorage('/tmp')

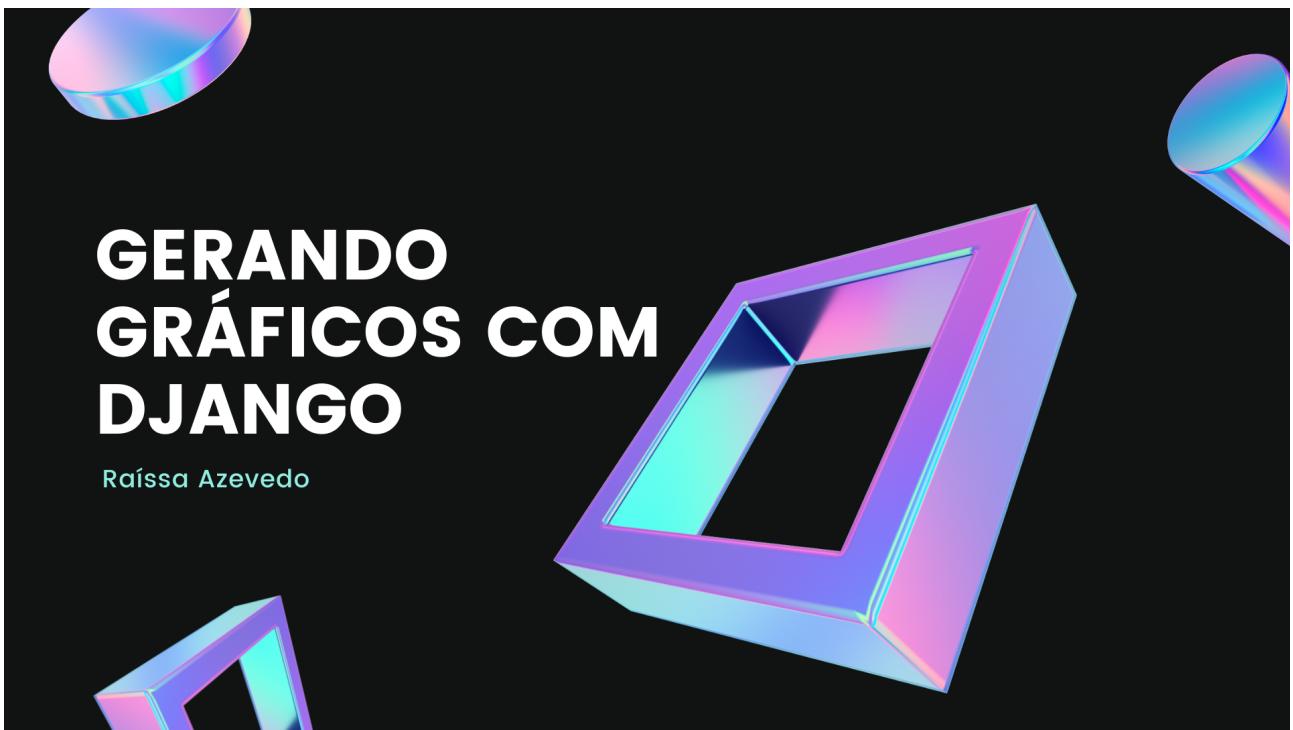
        with fs.open('relatorio2.pdf') as pdf:
            response = HttpResponseRedirect(pdf, content_type='application/pdf')
            # Faz o download do arquivo PDF
            #response['Content-Disposition'] = 'attachment; filename="relatorio2.pdf"'
            # Abre o PDF como página HTML:
            response['Content-Disposition'] = 'inline; filename="relatorio2.pdf"'
        return response
```

Na urls da aplicação:

```
from django.urls import path
from .views import IndexView, Index2View

urlpatterns = [
    path('', IndexView.as_view(), name='index'),
    path('2/', Index2View.as_view(), name='index2'),
]
```

Essas são as duas melhores formas de gerar arquivo PDF com Django.



GERANDO GRÁFICOS COM DJANGO

Raíssa Azevedo

[Raíssa Azevedo](#)

Como gerar gráficos utilizando Django?

Usando recursos de uma biblioteca chamada **Chart.js**.

O novo projeto será: *charts*

```
pip3 install django django-chartjs django-bootstrap
django-admin startproject charts .
django-admin startapp core
```

Fazer todas as configurações necessárias do `Settings.py`.

Criar um arquivo de rotas (`urls`) na aplicação `core`:

```
from django.urls import path
from .views import IndexView, DadosJSONView

urlpatterns = [
    path('index', IndexView.as_view(), name='index'),
    path('dados/', DadosJSONView.as_view(), name='dados'),
]
```

Na url do Projeto, indicar as rotas criadas:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('', include('core.urls')),
    path('admin/', admin.site.urls),
]
```

Será necessária a criação dos templates:

```
{% load static %}
{% load bootstrap4 %}

<!doctype html>
<html lang="pt-br">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    {% bootstrap_css %}

    <title>Charts!</title>
</head>
<body>
<div class="container">
    <h1 class="text-primary">Charts!</h1>
</div>
<div class="container">
    <canvas id="grafico" width="500" height="400"></canvas>
</div>
    <!-- Optional JavaScript -->
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->
    {% bootstrap_javascript jquery='full' %}

    <script type="text/javascript" src="{% static 'js/Chart.min.js' %}"></script>
    <script type="text/javascript">
        $.get('{% url 'dados' %}', function(data){
            var ctx = $('#grafico').get(0).getContext("2d");
            new Chart(ctx, {
                type: 'line', data: data
            });
        });
    </script>
</body>
</html>
```

Obs: Nesse projeto não se faz necessário a criação da pasta Static com os arquivos min.js, porque o Django entende que é pra buscar dentro da biblioteca Static indicada no inicio da página HTML.

No Views.py:

```
from random import randint
from django.views.generic import TemplateView
from chartjs.views.lines import BaseLineChartView

class IndexView(TemplateView):
    template_name = 'index.html'

class DadosJSONView(BaseLineChartView):

    def get_label(self):
        """Retorna 12 labels para a apresentação do x"""
        labels = [
            'Janeiro',
            'Fevereiro',
            'Março',
            'Abril',
            'Maio',
            'Junho',
            'Julho',
            'Agosto',
            'Setembro',
            'Outubro',
            'Novembro',
            'Dezembro'
        ]
        return labels

    def get_providers(self):
        """Retorna os nomes dos datasets"""
        datasets = [
            'Programação para Leigos',
            'Algoritmos e Lógica de Programação',
            'Programação em C#',
            'Programação em Python',
            'Banco de Dados'
        ]
        return datasets

    def get_data(self):
        """Retorna 6 datasets para plotar o Gráfico
        E cada linha representa um dataset,
        Cada coluna representa um label
        A quantidade de dados precisa ser igual aos datasets/labels"""
        dados = []
        for l in range(6):
            for c in range(12):
                dado = [
                    randint(1, 200), # Jan
                    randint(1, 200), # Fev
                    randint(1, 200), # Mar
                    randint(1, 200), # Abr
                    randint(1, 200), # Mai
                    randint(1, 200), # Jun
                    randint(1, 200), # Jul
                    randint(1, 200), # Ago
                    randint(1, 200), # Set
                    randint(1, 200), # Out
                    randint(1, 200), # Nov
                    randint(1, 200) # Dez
                ]
                dados.append(dado)
        return dados
```

O Randint irá gerar dados aleatórios de preenchimento para o gráfico. Gerando um número inteiro de 1 a 199.

Em seguida, executar o migrate:

```
python3 manage.py migrate
```

Em seguida, fazer as configurações para publicação:

```
heroku login  
git init  
git status  
git add .  
git commit -m 'Projeto Finalizado'
```



[Raíssa Azevedo](#)

```
pip3 install django social-auth-app-django django-bootstrap4
```

Após instalar o Django e fazer as configurações básicas...

No settings.py indicar as aplicações básicas instaladas...

```
'core',
'bootstrap4',
'social_django',
```

As autentificações de facebook são feitas dessa forma.

```
AUTHENTICATION_BACKENDS = [
    'social_core.backends.facebook.FacebookOAuth2',
    # Apenas se quiser manter a autentificação padrão do django
    'django.contrib.auth.backends.ModelBackend',
]
```

Todas as autentificações de social medias seguem a mesma premissa.

Os tipos de autentificações suportadas por python e django podem ser encontrados em:

[CLIQUE AQUI](#)

Ou pesquise no google **Python Social Auth** em seguida de backends.

Em templates(settings.py), adicionar mais 2 context processors:

```
'social_django.context_processors.backends',
'social_django.context_processors.login_redirect',
```

Depois, definir as rotas pra quando houver Login e o Logout da página.

```
LOGIN_URL = 'login'
LOGIN_REDIRECT_URL = 'index'
LOGOUT_URL = 'logout'
LOGOUT_REDIRECT_URL = 'login'
```

Em seguida, elaborar as rotas nas URLS. Na url do projeto, a configuração é padrão.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("", include('core.urls')),
    path('admin/', admin.site.urls),
]
```

Entretanto na rota do aplicativo, a configuração muda um pouco.

```
from django.urls import path, include
from django.contrib.auth import views as auth_views

from core.views import IndexView, LoginView

urlpatterns = [
    path('login/', LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('social-auth/', include('social_django.urls', namespace='social')),
    path("", IndexView.as_view(), name='index'),
]
```

Isso ocorre devido as configurações de backend para a conexão de login com redes sociais não serem padrões do django, entretanto, o framework possui algumas configurações facilitadoras.

Na views.py:

```
from django.views.generic import TemplateView
from django.contrib.auth.mixins import LoginRequiredMixin

class IndexView(LoginRequiredMixin, TemplateView):
    template_name = 'index.html'

class LoginView(TemplateView):
    template_name = 'login.html'
```

Os Templates:

A base.html:

```
{% load static %}
{% load bootstrap4 %}
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device.width, initial-scale=1.0"/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge"/>
    {% bootstrap_css %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}" />
    <title>Django Social</title>
</head>
<body>
    <div class="container-fluid">
        <div>
            <h1 class="text-white text-center">{% block title %}{% endblock %}</h1>
            <div class="card p-5">
                {% block content %}{% endblock %}
            </div>
        </div>
    </div>
    {% bootstrap_javascript jquery='full' %}
</body>
</html>
```

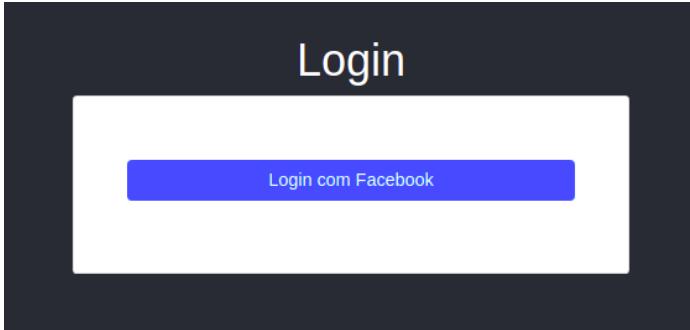
O index:

```
{% extends 'base.html' %}
{% load static %}
{% block title %}Inicio{% endblock %}
{% block content %}
    <div class="row">
        <div class="col-sm-12 mb-3">
            <h4 class="text-center">Bem vindo(a) {{ user.username }}</h4>
        </div>
    </div>
{% endblock %}
```

O login.html:

```
{% extends 'base.html' %}
{% load static %}
{% block title %}Login{% endblock %}
{% block content %}
    <div class="row">
        <div class="col-md-8 mx-auto social-container my-2 order-md-1">
            <button class="btn btn-primary mb-2">
                <a href="#">Login com Facebook</a>
            </button>
        </div>
    </div>
{% endblock %}
```

Isso irá gerar uma página de login padrão:



Em seguida, acessar as configurações de developers do Facebook, para ter acesso as chaves.

E criar o aplicativo para acesso pelo instagram.

Nas configurações básicas do aplicativo do facebook, denominar os dominios (localhost, ou onde o site tiver hospedado).

Depois voltar o settings da IDE e fazer as seguintes configurações:

```
# Configurações para qualquer rede social
SOCIAL_AUTH_RAISE_EXCEPTIONS = False
```

Se a pessoa já tiver feito acesso pelo facebook, deslogado, e em outro dia logar novamente, a conta já está salva, e isso impede a criação de uma exceção.

```
# Configurações para Facebook
SOCIAL_AUTH_FACEBOOK_KEY =
SOCIAL_AUTH_FACEBOOK_SECRET =
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email', 'user', 'link']
SOCIAL_AUTH_FACEBOOK_PROFILE_EXTRA_PARAMS = {
    'fields': 'id, name, email, picture.type(large), link'
}
SOCIAL_AUTH_FACEBOOK_EXTRA_DATA = [
    ('name', 'name'),
    ('email', 'email'),
    ('picture', 'picture'),
    ('link', 'profile_url'),
]
```

Qualquer seja a configuração para a criação de login através da rede social. Irá gerar essas duas chaves (Secret - Key).

Em seguida voltar ao facebook developers.

Copiar o ID do aplicativo colar em **Key** e copiar a chave secreta do aplicativo e colar em **secret**.

Em seguida, voltar ao login html e corrigir o endereço href.

```
<a href="{% url 'social:begin' 'facebook' %}>Login com Facebook</a>
```

A configuração é a mesma para qualquer rede social.

Fazer uma ultima modificação no index.

```
{% extends 'base.html' %}  
{% load static %}  
{% block title %}Inicio{% endblock %}  
{% block content %}  
    <div class="row">  
        <div class="col-sm-12 mb-3">  
            <h4 class="text-center">Bem vindo(a) {{ user.username }}</h4>  
        </div>  
        {% for a in backends.associated %}  
            {% if a.provider == 'facebook' %}  
                <div class="col-md-4 text-center">  
                      
                </div>  
                <div class="col-md-8 social-container my-2">  
                    <p>Logado via: {{ a.provider|title }}</p>  
                    <p>Nome: {{ a.extra_data.name }}</p>  
                    <p>Profile:<a href="{{ a.extra_data.profile_url }}">Link</a></p>  
                </div>  
            {% endif %}  
        {% endfor %}  
        <div class="col-sm-12 mt-2 text-center">  
            <button class="btn btn-warning">  
                <a href="{% url 'logout' %}">Logout</a>  
            </button>  
        </div>  
    </div>  
{% endblock %}
```



MELHORANDO A SEGURANÇA DA APLICAÇÃO DJANGO

Raíssa Azevedo

[Raíssa Azevedo](#)

Como melhorar a segurança da aplicação Django?

Neste projeto não se faz necessária a criação de um aplicativo, somente do projeto.

```
django-admin startproject segurança .
```

O próprio arquivo de **settings** do projeto já possui dados de segurança, como o **secret key** que precisam ser confidenciais.

Fazer as configurações básicas do Settings.py.

O Django já possui por padrão várias configurações de segurança, como: Secret_key, Middleware security, auth validations.

RECURSOS AUTOMÁTICOS DE SEGURANÇA DO DJANGO:

- **Cross Site Scripting (XSS):**

Que permite o atacante a injetarem scripts em sites, com HTML ou JavaScript. Mudando o comportamento das páginas Web.

- **Cross Site Request Forgery (CSRF)**

{% csrf_token %}

Recurso que impede a exploração de um website que transições não permitidas sejam relacionadas.

Porque toda entrada de usuário pode ser perigosa e precisa ser validada.

O Django provê um token nos formulários.

- **SQL Injection:**

O django vem preparado contra SQL Injection, é um dos ataques mais comuns, através de um formulário de entrada de dados, e quando clica no botão para fazer login e ocorre a combinação do dado fornecido com o salvo no banco de dados. Sites sem essa proteção, o cracker insere comandos que vão modificar o comportamento do que está sendo escrito, podendo até conceder permissão de adm.

- **Suporta HTTPS e TLS:**

Quando a aplicação é startada com o *runserver* entra em HTTP (Protocolo da internet sem segurança) e não no HTTPS, ele permite que a aplicação só funcione em HTTPS.

- **Armazenamento Seguro de Senhas**

Qualquer senha nunca deve ser armazenada em banco de dados em texto puro, ela precisa ser criptografada.

- **Algoritmo PBKDF2 com hash SHA256 recomendado pelo Instituto NIST:**

É o algoritmo de criptografia de dados utilizados pelo Django.

Por padrão o acesso a área administrativa do django é feito através do **/admin**

Uma coisa que pode ser feita, é mudar o nome do local da administração, por Painel, controle ou outra coisa.:

NaUrls.py do projeto

```
urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Trocar a rota admin, para outro nome.

Outra coisa que pode ser feita, é habilitar novos recursos de segurança, como:

```
# RECURSOS EXTRAS DE SEGURANÇA DO DJANGO
SECURE_HSTS_SECONDS = True
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_BROWSER_XSS_FILTER = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
CSRF_COOKIE_HTTPONLY = True
X_FRAME_OPTIONS = 'DENY'

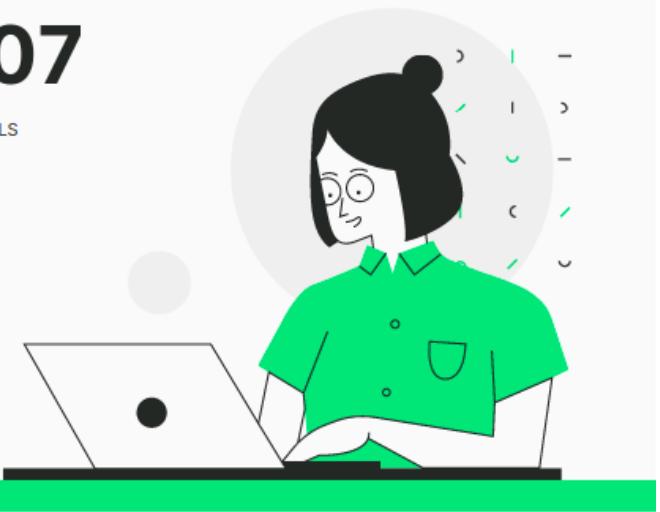
# SECURE_SSL_REDIRECT = True
```

O ideal é ativar o ultimo recurso somente no momento da aplicação, ou a aplicação não irá rodar localmente no momento do desenvolvimento.

SEÇÃO 07

DISSECANDO O DJANGO USER MODELS

07



SEÇÃO 07 – DISSECANDO O DJANGO USER MODEL

Entendendo o Django User Model:

Quando uma aplicação Django é criada, tem todo um sistema de manutenção que é criado automaticamente pelo Framework. Entretanto, esse modelo pode ser integrado a outros, ou modificado.

O projeto irá se chamar “Django User Model”

```
pip3 install django  
django-admin startproject djangousermodel .
```

Para entender o funcionamento do User Model:

```
python manage.py shell  
python manage.py migrate
```

Fazendo uso do Django Shell:

```
from django.contrib.auth.models import User
```

O que é o Model User?

É o módulo models e dentro desse modo é importado os modelo de dados. Ou seja, é o modelo de dados.

dir(User) ou **help(User)** para saber o que pode ser usado, ou olhar a documentação necessária.

```
help(manageruser.create)
```

Pode usar o método create user ou create superuser.

Criando um usuário:

```
>>> usuario = User.objects.create_user(username='teste', password='123456', email='test@gmail.com')
>>> usuario
<User: teste>
>>> usuario.save()
```

Como consultar um usuário:

```
ret = User.objects.all()
ret
<QuerySet [<User: teste>]>
>>> ret[0].username
'teste'
>>> ret[0].password
'pbkdf2_sha256$260000$r1nvLTJn6YnwHkpOcCWUhd$YQj kWMvriWB3J+5LWAkmB3kMwpap4SaOjs2NQrt2Vs='
```

OBS: Ou seja, o Django criptografou a senha que era “123456”.

Se criptografar o 123456 com Django fica:

```
'pbkdf2_sha256$260000$r1nvLTJn6YnwHkpOcCWUhd$YQj/kWMvriWB3J+5LWAkmB3kMwpap4SaOjs2NQrt2Vs
='
```

Um Staff pode logar na área administrativa, mas não pode visualizar ou modificar nada. É preciso criar um superuser. Ou seja, um usuário pode ser staff, mas não necessariamente é um super usuário.

ESTENDENDO O DJANGO USER MODELS

O projeto irá se chamar Django User Model 2

```
django-admin startproject djangousermodel2 .
Django-admin startapp core
```

Fazer todas as configurações básicas do settings.py

Criar um models na aplicação core

Isso inclui:

- Permitir todos usuários [‘*’]
- Indicar o novo app: ‘core’
- Mudar o Banco de Dados se necessário
- Mudar a região “America/Sao_Paulo”
- E fazer os Medias e Statics Root

Existem 3 formas de fazer a aplicação para integração:

1^a Forma:

```
from django.contrib.auth.models import User

class Post(models.Model):
    autor = models.ForeignKey(User, verbose_name='Autor', on_delete=models.CASCADE)
    titulo = models.CharField('Titulo', max_length=100)
    texto = models.TextField('Texto', max_length=400)

    def __str__(self):
        return self.titulo
```

O User presente nessa forma, é o próprio do Django o que dificulta a customização da página administrativa.

2^a Forma:

A 2^a forma é utilizada quando fazemos a customização da área administrativa do Web Site:

Primeiro há a necessidade de fazer algumas modificações no settings.py:

```
AUTH_USER_MODEL = 'usuarios.CustomUsuario'
```

Ou seja, ocorre a substituição do módulo de usuário padrão para o criado customizado.

E no arquivo models:

```
from django.db import models
from django.conf import settings

class Post(models.Model):
    autor = models.ForeignKey(settings.AUTH_USER_MODEL, verbose_name='Autor',
on_delete=models.CASCADE)
    titulo = models.CharField('Titulo', max_length=100)
    texto = models.TextField('Texto', max_length=400)

    def __str__(self):
        return self.titulo
```

Ou seja, precisa haver a indicação de mudança para o padrão customizado indicado anteriormente no arquivo de settings.

3ª Forma:

Obs: É a forma mais indicada para introduzir uma área administrativa customizada.

O módulo de autenticação do Django tem um get_user_model para a executar a função customizada. Se tiver sobrescrito, irá trazer a forma customizada, caso contrário ele irá exibir a área administrativa oficial.

```
from django.db import models

from django.contrib.auth import get_user_model

class Post(models.Model):
    autor = models.ForeignKey(get_user_model(), verbose_name='Autor', on_delete=models.CASCADE)
    titulo = models.CharField('Título', max_length=100)
    texto = models.TextField('Texto', max_length=400)

    def __str__(self):
        return self.titulo
```

Ao acessar o módulo com `python manage.py runserver`

O modulo irá ser exibido.

Registrar no Admin.py o modelo modificado criado em models,

```
from django.contrib import admin

from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('titulo', 'autor')
```

Como fazer para exibir nome e sobrenome na área administrativa:

Existe um método em user chamado “get_full_name”

Então:

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('titulo', '_autor')

    def _autor(self, instance):
        return f'{instance.autor.get_full_name()}'
```

Depois dessas alterações, na área administrativa irá aparecer o Nome Completo ao invés do nome de usuário utilizado para fazer login.

Selecione post para modificar

ADICIONAR POST +

Ação:	
0 de 2 selecionados	
<input type="checkbox"/>	TÍTULO
<input type="checkbox"/>	Post da Felicity
<input type="checkbox"/>	Post 1 do Udemy
AUTOR	
	Felicity Jones
	Raissa Azevedo
2 posts	

Imagina que há um sistema com vários usuários, e você não quer que os demais tenham acesso aos dados dos demais usuários.

Como fazer para cada usuário observar somente os próprios Posts?

```
def get_queryset(self, request):
    qs = super(PostAdmin, self).get_queryset(request)
    return qs.filter(autor=request.user)
```

Essa modificação é feita no Admin.py e ela entende que é pra mostrar somente os dados do usuário que está logado, tudo relacionado a outro usuário, fica oculto.

Para ter acesso ao model:

Pressione CTRL e clique no model.

Como pegar somente o dado da sessão que tá logado e ocultar a exibição de usuários na hora de fazer um novo Post.

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('titulo', '_autor')
    exclude = ['autor', ]
```

Isso irá excluir a lista de autores que aparecia no módulo administrativo antes. Porém haverá um erro na hora de salvar o novo post. Portanto é preciso fazer uma alteração no modo salvar do Post.

```
def save_model(self, request, obj, form, change):
    obj.autor = request.user
    super().save_model(request, obj, form, change)
```

No admin.py essa alteração significa que a pessoa que está logada é a autora do Post. Dessa forma, o erro de salvar para de ocorrer.

Como alterar o textos que indicam o nome da parte de administração:

No arquivo de URLs do Projeto:

```
urlpatterns = [
    path('admin/', admin.site.urls),
]

admin.site.site_header = 'Geek University'
admin.site.site_title = 'Evolua seu lado geek!'
admin.site.index_title = 'Sistema de Gerenciamento de Posts'
```

Todas as alterações serão mostradas na sessão administrativa.

CRIANDO UM USER MODEL CUSTOMIZADO

O novo projeto de modificação de área Administrativa irá se chamar “Django User Model 3”

```
django-admin startproject djangousermodel3 .
django-admin startapp usuarios
```

A aplicação desse projeto é chamada de usuários.

Para logar em um site, o django pede nome de usuário e senha, isso não é interessante, portanto o indicado é pedir e-mail e senha. E em outros casos pode ser CPF e senha ou telefone e senha.

Aqui entra o Model User customizado.

Existem 2 formas de criar um usuário customizado:

1ª Forma:

```
from django.db import models

from django.contrib.auth.models import AbstractBaseUser

class CustomUsuario(AbstractBaseUser):
    pass

    def __str__(self):
        return self.email
```

2ª Forma:

Todo o procedimento idêntico, entretanto, utilizando “AbstractUser”

Qual a diferença de AbstractBaseUser para AbstractUser?

Ambos tratam de usuário abstrato. Entretanto, o Base é um usuário mais básico. Ou seja, faltam funcionalidades, que precisam ser implementadas manualmente.

Obs: Portanto o recomendado é utilizar o AbstractUser

Será necessário utilizar um Gerenciador de Usuário:

No arquivo models.py:

```
from django.db import models
from django.contrib.auth.models import AbstractUser, BaseUserManager

class UsuarioManager(BaseUserManager):
    use_in_migrations = True

    def _create_user(self, email, password, **extra_fields):
        if not email:
            raise ValueError('O e-mail é obrigatório')
        email = self.normalize_email(email)
        user = self.model(email=email, username=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_user(self, email, password=None, **extra_fields):
        # extra_fields.setdefault('is_staff', True) O recomendado é deixar como False, padrão do Django
        extra_fields.setdefault('is_superuser', False)
        return self._create_user(email, password, **extra_fields)

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_superuser', True)
        extra_fields.setdefault('is_staff', True)

        if extra_fields.get('is_superuser') is not True:
            raise ValueError('Superuser precisa ter is_superuser=True')

        if extra_fields.get('is_staff') is not True:
            raise ValueError('Superuser precisa ter is_staff=True')

        return self._create_user(email, password, **extra_fields)

class CustomUsuario(AbstractUser):
    email = models.EmailField('E-mail', unique=True)
    fone = models.CharField('Telefone', max_length=15)
    is_staff = models.BooleanField('Membro da Equipe', default=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name', 'fone']

    def __str__(self):
        return self.email

    objects = UsuarioManager()
```

No `create_user` que indica o usuário padrão, o recomendado é deixar o `setUp` de `staff` como `False`. E deixar como `True` somente nas configurações de Super User que indica o usuário administrador.

O campo `objects=` no final do código: Indica que todo o processo vai ser gerenciado pelo `UsuarioManager`. Finalizando com o `models.py`, é preciso criar dois formulários para preenchimento da sessão de usuário.

Dentro do diretório da aplicação usuários criar um arquivo **forms.py**

No forms.py:

Serão criados os formulários de criação do usuário (UserCreateForm) e de alteração de usuário (UserChangeForm)

```
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUsuario
```

```
class CustomUsuariocreateForm(UserCreationForm):
```

```
    class Meta:
        model = CustomUsuario
        fields = ['first_name', 'last_name', 'fone']
        labels = {'username': 'Username/E-mail'}

    def save(self, commit=True):
        user = super().save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        user.email = self.cleaned_data["username"]
        if commit:
            user.save()
        return user
```

```
class CustomUsuariochangeform(UserChangeForm):
```

```
    class Meta:
        model = CustomUsuario
        fields = ['first_name', 'last_name', 'fone']
```

Esses são os dois formulários necessários, um para criação e outro que permite alterar os dados.

Em seguida, registrar todos os models no Admin.py:

Fazer o import de tudo que for necessário para a administração do usuário, uma vez que é na página de administração que as alterações estão sendo feitas.

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin

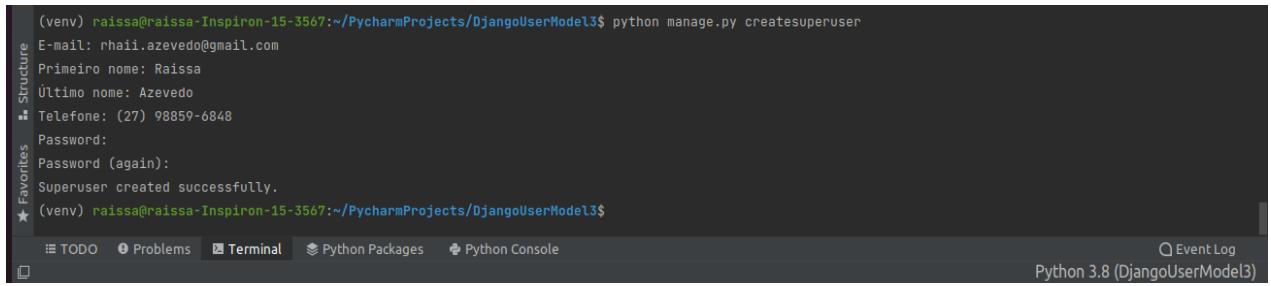
from .forms import CustomUsuariocreateForm, CustomUsuariochangeform
from .models import CustomUsuario
```

```
@admin.register(CustomUsuario)
class CustomUsuarioAdmin(UserAdmin):
    add_form = CustomUsuariocreateForm
    form = CustomUsuariochangeform
    model = CustomUsuario
    list_display = ['first_name', 'last_name', 'fone', 'is_staff']
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        ('Informações Pessoais', {'fields': ('first_name', 'last_name', 'fone')}),
        ('Permissões', {'fields': ('is_active', 'is_staff', 'is_superuser', 'groups', 'user_permissions')}),
        ('Datas Importantes', {'fields': ('last_login', 'date_joined')}),
```

```
)
```

O **fieldsets** são todas as configurações padrões do Django que pedem os dados de cadastro do usuário. Há outros dados que podem ser acrescentados.

As mudanças já ocorrem dentro do próprio terminal:



```
(venv) raissa@raissa-Inspiron-15-3567:~/PycharmProjects/DjangoUserModel3$ python manage.py createsuperuser
E-mail: rhaii.azevedo@gmail.com
Primeiro nome: Raissa
Último nome: Azevedo
■ Telefone: (27) 98859-6848
Password:
Password (again):
Superuser created successfully.
★ (venv) raissa@raissa-Inspiron-15-3567:~/PycharmProjects/DjangoUserModel3$
```

The screenshot shows a PyCharm interface with a terminal window open. The terminal displays the command `python manage.py createsuperuser` being run, followed by prompts for email, first name, last name, phone number, and password. It then confirms that a superuser was created successfully. The PyCharm interface includes a sidebar with 'Favorites' and 'Structure' sections, and a bottom navigation bar with tabs for 'TODO', 'Problems', 'Terminal', 'Python Packages', 'Python Console', and 'Event Log'. The status bar at the bottom right indicates 'Python 3.8 (DjangoUserModel3)'.

LOGIN E AUTENTICAÇÃO

Quer uma nova tela de autenticação, que seja diferente da oferecida pelo Django. O projeto utilizado será o mesmo.

Ir na parte de URLs:

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic.base import TemplateView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('contas/', include('django.contrib.auth.urls')),
    path('', TemplateView.as_view(template_name='index.html'), name='index'),
]
```

Em seguida é preciso criar uma estrutura de templates para as telas de login. Dessa vez, o diretório de templates precisa ser criado na raiz do projeto, ao invés da aplicação como nos projetos anteriores. Porque o que vai ser modificado é a parte administrativa, e não o site.

Então, criar um diretório de templates na raiz do projeto.

Criar um arquivo html denominado **base.html** e fazer as modificações necessárias.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Geek University</title>
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit-no">
    <link rel="icon" href="https://www.geekuniversity.com.br/static/images/favicon.4fcb819d32bf.ico">
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" rel="stylesheet"
id="bootstrap">
    <link href="https://getbootstrap.com/docs/4.0/examples/sign-in/signin.css" rel="stylesheet">
</head>
<body class="text-center">
    {% block content %}
    {% endblock %}
    <script src="http://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/jquery@3.2.1/jquery.min.js"></script>
</body>
</html>
```

Garantir ter adicionado o favicon, o css e o Javascript. Por se tratar da página base. Adicionar os blocos de conteúdo no body padrões do Django.

Dentro do diretório templates, criar o **index.html**

Nas configurações da pagina “index” indicar que se o usuário for anônimo colocar uma área para Login
E se o usuário tiver autenticado exibir a mensagem de “Bem Vindo” e o botão de logoff

```
{% extends 'base.html' %}  
{% block content %}  
    <div class="container">  
        <h1>Geek University</h1>  
        {% if user.is_anonymous %}  
            <a class="btn btn-primary" href="{% url 'login' %}">Login</a>  
        {% else %}  
            <div class="alert alert-primary" role="alert">  
                Seja bem vindo(a), {{ user.get_full_name }}!  
            </div>  
            <a class="btn btn-primary" href="{% url 'logout' %}">Logout</a>  
        {% endif %}  
    </div>  
{% endblock %}
```

Por fim, é necessário criar a página de Login:

Dentro do diretório de templates, criar o diretório “registration” criar o arquivo **login.html**

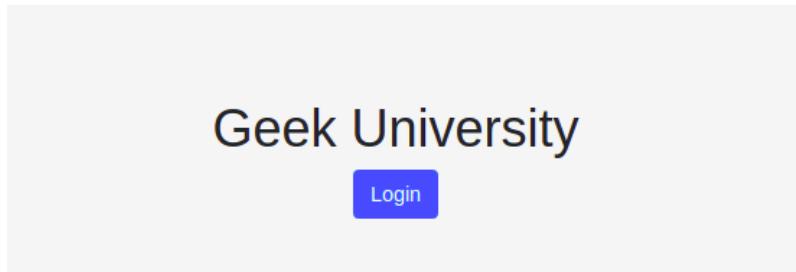
```
{% extends 'base.html' %}  
{% block content %}  
    <form class="form-signin" method="post" autocomplete="off">  
        {% csrf_token %}  
          
        <h1 class="h3 mb-3 font-weight-normal">Informe seus dados</h1>  
  
        <label for="username" class="sr-only">E-mail</label>  
        <input type="email" id="username" name="username" class="form-control" placeholder="Informe seu e-mail"  
required autofocus>  
  
        <label for="password" class="sr-only">Senha</label>  
        <input type="password" id="password" name="password" class="form-control" placeholder="Informe sua senha"  
required>  
  
        <button class="btn btn-lg btn-primary btn-block" type="submit">Acessar</button>  
        <p class="mt-5 mb-3 text-muted">&copy; {{ now|date:'Y' }}</p>  
    </form>  
{% endblock %}
```

Fazer os ajustes de redirecionamento de login no settings.py

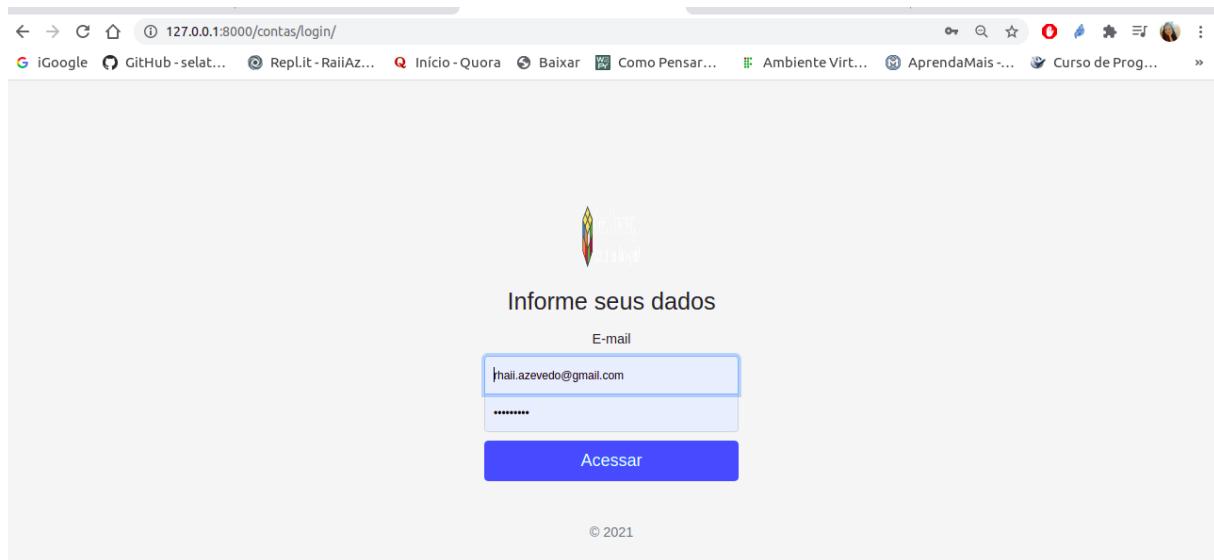
```
LOGIN_REDIRECT_URL = 'index'  
LOGOUT_REDIRECT_URL = 'index'
```

Essas configurações indicarão que volte para a pagina principal automaticamente.

Acessando o resultado das configurações de Página administrativa customizada:



Irá pedir o Login:



E depois irá exibir a página de Boas Vindas como indicado.

