

Desenvolvimento de Aplicações Web Responsivas com Django 4 e Bootstrap 5

Seja muito bem-vindo ao curso Desenvolvimento de Aplicações Web Responsivas com Django 4 e Bootstrap 5. Primeiramente, eu gostaria agradecer pela aquisição deste curso e pela confiança depositada em nós. Ao mesmo tempo, eu gostaria de te assegurar que você tem em mãos um material que foi cuidadosamente pensado para enriquecer suas experiências de aprendizado.

Trabalharemos nossa capacidade de abstração e generalização, resolvendo problemas complexos a partir de tarefas simples, sempre em ordem crescente de dificuldade. Tudo isso para que você tenha autonomia para resolver problemas e construir suas próprias aplicações.

E não é só isso! Todo o código-fonte desenvolvido estará disponível no Github para que você possa consultar sempre que precisar. Todos os commits foram pensados para facilitar a comparação entre o código-fonte da solução esperada com o código feito por você. Com isso, você poderá identificar rapidamente as diferenças entre códigos-fonte, e perguntas como: “O que é que estou fazendo de errado?” serão respondidas o mais rápido possível.

E não é só isso! Você receberá um PDF contendo resumos dos principais tópicos abordados na aula e o passo-a-passo de tudo o que fizemos. O conteúdo desse material terá links para a documentação oficial, caso você queira se aprofundar ainda mais.

E o que iremos fazer durante o curso? A gente vai desenvolver o blog da Academia Python. Não se engane, embora a ideia pareça simples, iremos cobrir tópicos básicos e avançados do Django e do Bootstrap.

Sumário

Preparando o ambiente linux	5
Instalando o Python	5
Instalando o pip	6
Instalando o Git	6
Instalando o Pycharm	6
Instalando o PostgreSQL	6
Instalando o Dbeaver	8
django framework - principais características	9
Origens do framework	9
Quem utiliza Django?	9
Principais características	10
Arquitetura do Django	10
Hello, World! Django	11
Criando o projeto	11
Entendendo a estrutura de um projeto Django	12
Criando a primeira aplicação Django.	13
Entendendo a estrutura de uma aplicação Django	15
Fluxo de requisições django – Parte I	15
Um pouco de git faz bem!	16
O que é o Git?	16
Git Workflow	17
Principais comandos	17
Primeiro model	19
Criando o primeiro Model	19
Configurando o Postgres	19
Migrations	20
Introdução ao Django Admin	20
CRIANDO NOSSO PRIMEIRO TEMPLATE	21
Conhecendo um pouco da API QuerySet	22
CRUD com a API QuerySet.	22
Criando objetos	22
6. Abra o DBeaver para confirmar a criação do registro na base de dados.	22

Recuperando objetos	22
Editando objetos	23
Deletando objetos	23
listando posts cadastrados	24
navegando entre páginas	24
Configurando os comandos runserver, makemigrations e migrate no PyCharm	26
Configurações do comando python manage.py runserver	26
Configuração do comando python manage.py makemigrations	27
Configuração do comando python manage.py migrate	28
Fluxo de requisições - Parte II	28
Desmistificando os static files	29
Configurando editor de textos similar ao Wordpress (CKEditor)	30
INTENSIVÃO BOOTSTRAP	32
INSTALAÇÃO DO BOOTSTRAP	32
CONTAINERS	33
ENTENDENDO O CONCEITO DE MOBILE-FIRST	33
MOBILE FIRST E CONTAINER BREAKPOINTS	34
SISTEMA DE GRIDS	35
NAVBAR	35
FLEXBOX	36
CARD	37
GRID CARDS	37
Formulários com bootstrap e django + encaminhamento de emails	37
Formulários - Parte I. Criando e validando formulários com Bootstrap e Django.	38
Formulários - Parte II. Criando e validando formulários com a classe Forms do Django.	39
Form - Primeiros passos	39
Personalizado CSS do formulário	40
Formulários - Parte III. Criando formulários com a classe ModelForm	41
ModelForm - Primeiros passos	42
Sobre o encaminhamento de emails	44
Configurando servidor SMTP do Gmail	44
Configurando o settings.py para encaminhamento de emails utilizando SMTP do Gmail Para que os emails sejam encaminhados pelo Django, precisamos acrescentar algumas informações de configuração no arquivo settings.py.	47
IMPLEMENTANDO SISTEMA DE AUTENTICAÇÃO CONTROLE DE ACESSO (em construção)	48

Realizando cadastro de usuários	49
Validando preenchimento dos campos com HTMX	51
Login com Redes Sociais	53
Extends User Model, Managers e Signals	55
Desafio de programação (Projeto base da seção)	55
Managers	55
Alterando o nome do manager	56
Criando Custom Manager	56
Utilizando Custom Manager	56
Utilizando manager renomeado	56
Utilizando manager personalizado	56
Extends User Model	57
Proxy Model	57

PREPARANDO O AMBIENTE LINUX

Instalando o Python

Por padrão, o Python já vem instalado nas distribuições Linux. Antes de iniciar o processo de instalação, certifique-se de que ele não está instalado em sua estação de trabalho utilizando o comando listado no **passo VI** deste tutorial. Caso você deseje instalar a versão do Python 3.10 ou versão mais recente, siga os passos abaixo.

- I. Baixe as atualizações e informações de todos os repositórios catalogados em **etc/apt/sources.list**.

» **sudo apt update**

- II. Instale as versões mais recentes dos pacotes instalados em seu sistema.

» **sudo apt upgrade**

- III. Instale o repositório Deadsnakes.

» **sudo add-apt-repository ppa:deadsnakes/ppa**

- IV. Instale o repositório Deadsnakes.

» **sudo apt install python3.10**

- V. Cheque se o python foi realmente instalado.

» **python3 --version**

- VI. Adicione a versão do Python que se encontra instalada na sua máquina juntamente com a versão 3.10 ao update-alternatives.

» **sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.10**

» **sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.9 2**

- VII. Atualize a versão do padrão do Python.

» **sudo update-alternatives --config python3**

- VIII. O Linux irá lhe perguntar qual versão do Python será a principal. Informe o número sequencial correspondente à versão 3.10.

- IX. Verifique se a versão padrão do Python foi atualizada.

Instalando o pip

O **pip (Python Package Index - PyPI)** é o software utilizado para instalar e gerenciar pacotes de software escritos em linguagem Python. A maioria das distribuições do Python vem com o pip pré-instalado. Sua instalação é extremamente simples, conforme podemos observar nos passos a seguir.

1. Certifique-se que o pip não está instalado em seu Sistema Operacional.

» **pip --version**

2. Instale o pip, caso ele não esteja instalado.

» **sudo apt install pip**

Instalando o Git

Assim como o Python, o Git já vem pré-instalado nos sistemas Linux. Para instalá-lo, é bem simples.

1. Certifique-se que o Git não está instalado, antes de prosseguir.

» **git --version.**

2. Instale o Git, caso ele não esteja presente em seu sistema.

» **sudo apt install git**

Instalando o Pycharm

A instalação do Pycharm é bem simples, dado que a JetBrains disponibilizou sua instalação via linha de comando, conforme podemos ver no [site da JetBrains](#).

» **sudo snap install pycharm-community --classic**

Instalando o PostgreSQL

Faremos a instalação do PostgreSQL seguindo o passo a passo indicado na [documentação oficial](#). **A partir do item V, temos comandos que são compatíveis entre os sistemas Windows e Linux.**

- I. Crie o arquivo de configuração do repositório.

» **sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt \$(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'**

Alguns detalhes sobre o comando acima:

- a) O trecho **sudo sh -c** permite que executemos um comando comandos entre aspas simples.
- b) **echo "deb http://apt.postgresql.org/pub/repos/apt ...** o comando echo imprime a string delimitada entre aspas simples ou duplas em um arquivo ou na tela do dispositivo.
- c) **... \$(lsb_release -cs)-pgdg main"**. O **\$(...)** é utilizado interpolar o retorno de um comando dentro de uma string.
- d) **lsb_release -cs** retorna o codinome da versão do Ubuntu. Podemos visualizar todas as informações de nosso release com o comando **cat /etc/os-release**. O comando **lsb_release -la** nos fornece todas as informações do codinome da versão instalada em nosso PC.
- e) O trecho **> /etc/apt/sources.list.d/pgdg.list** Cria o arquivo **pgdg.list** no path **/etc/apt/sources.list.d**, inserindo nele o conteúdo do comando posicionado à esquerda.
- f) Tente atualizar o **sources.list** com o comando **sudo apt update** e veja que você se deparará com um erro.

U

II. Adicione a signing key ao seu arquivo **etc/apt/sources.list**

» **wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -**

III. Baixe as atualizações e informações de todos os repositórios catalogados em **etc/apt/sources.list**.

» **sudo apt update**

IV. Instale a última versão do PostgreSQL.

» **sudo apt install postgresql**

V. Configure o usuário postgres.

- a) Logue no postgres como superusuário postgres(usuário padrão do SCBD).

» **sudo su postgres**

- b) Entre no modo prompt de comando do postgres

» **psql**

c) Visualize detalhes da conexão com o SGBD.

» **\conninfo**

d) Liste todos os bancos de dados existentes no postgres.

» **\l**

e) Volte para o prompt de comando do postgres.

» **:q**

f) Cadastre uma senha para ao postgres

» **\password postgres**

g) Faça logout do usuário postgres

» **\q**

h) saia do usuário root.

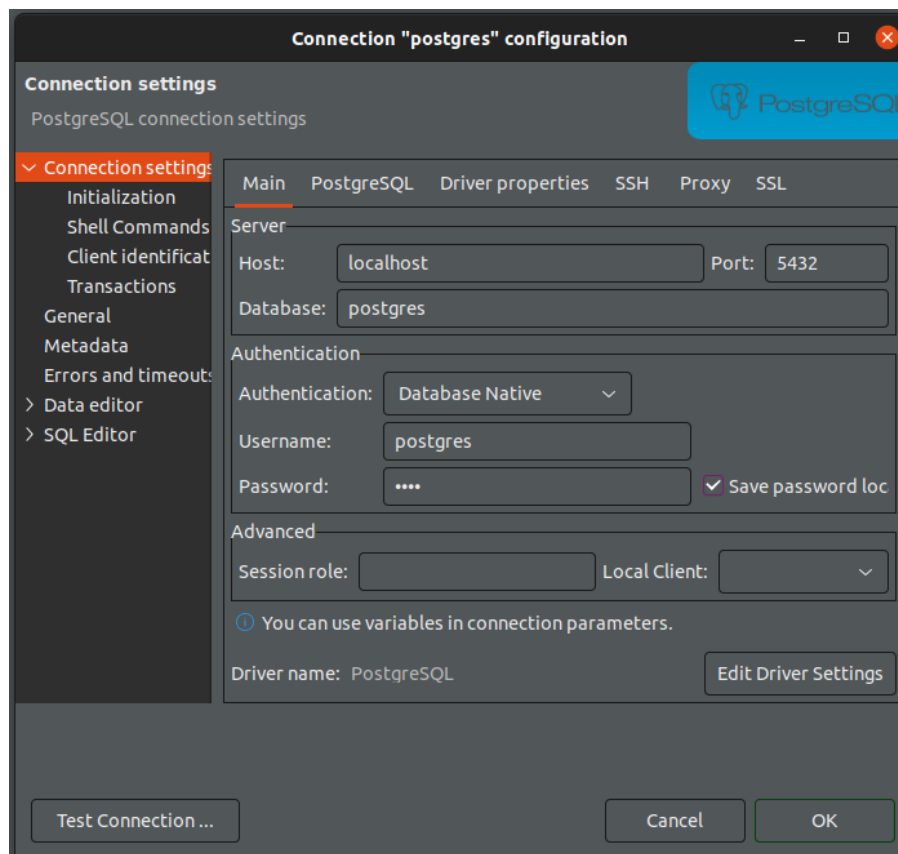
» **exit**

Instalando o Dbeaver

A instalação do Dbeaver é bastante simples, dado que podemos fazê-la utilizando o Snap, conforme consta no [site oficial](#).

» **sudo snap install dbeaver-ce**

Após a instalação do Dbeaver, você deve configurar a conexão com o banco de dados, seguindo as configurações apresentadas na figura abaixo.



DJANGO FRAMEWORK – PRINCIPAIS CARACTERÍSTICAS

Origens do framework

O Django é um framework para desenvolvimento ágil de projetos web escrito em Python. Ele foi criado para ser um sistema responsável pela gestão de um site jornalístico na cidade de Lawrence, no Kansas. Após isso, tornou-se um projeto de código aberto em 2005.

Curiosamente, o nome Django foi inspirado no músico de jazz Django Reinhardt, e não no filme Django Livre!

Quem utiliza Django?

Grandes players do mercado utilizam Django. Aqui vão alguns: **Mozilla, Instagram, Pinterest, National Geographic, Open Stack, Globo.com**, dentre outros.

Principais características

Aqui vão algumas características do framework.

- **Agilidade.** Quando dizemos que Django é ágil significa que você poderá desenvolver aplicações web de conceito em poucas horas!
- **DRY.** O Django utiliza o princípio **DRY** (Don't Repeat Yourself). Dizendo de outra forma: você não precisará reinventar a roda durante o desenvolvimento de projetos.
- **Reusabilidade.** As aplicações Django são “**Fully loaded**”. Isso significa que as aplicações poderão ser utilizadas em qualquer projeto.
- **Segurança.** O Django leva questões de segurança a sério. Ataques como SQL injection, cross-site scripting, cross-site request forgery and clickjacking recebem tratamento adequado já na fase de desenvolvimento.
- **Escalabilidade.** Projetos web escritos em Django têm a possibilidade de serem flexíveis e rapidamente escaláveis.
- **Versatilidade.** Com Django, podemos construir uma grande variedade de aplicações, desde redes sociais a aplicações de datascience.
- **ORM.** O Django possui um poderoso sistema de mapeamento objeto-relacional que nos permite modelar bancos de dados e consultar informações sem a necessidade de utilizarmos SQL.

Arquitetura do Django

O framework django estrutura seus projetos segundo o padrão MTV (Model, Template, View), onde:

- O **model** é responsável pelo mapeamento das classes Python para tabelas do banco de dados.
- O **template** representa a camada de visualização (HTML).
- A **view** é a camada que contém toda a lógica de negócio.

Essa arquitetura se assemelha ao MVC (Model-View-Controller), muito utilizada em frameworks como JSF, Zend e Laravel. O MTV tem semelhanças e diferenças com o MVC.

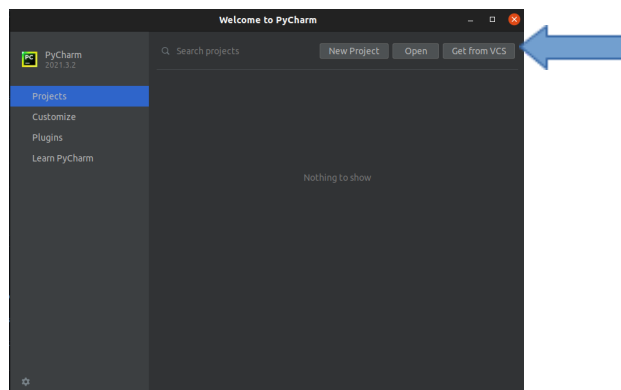
O model (M) assume as mesmas responsabilidades em ambas as arquiteturas, MTV e MVC. Já os Templates (T) do Django são equivalentes às Views (V) do MVC. Por fim, as Views (V) do Django possuem responsabilidades semelhantes às responsabilidades do Controller (C) MVC, mas são conceitualmente diferentes.

A equipe do Django entende que a camada view descreve quais dados serão apresentados ao usuário, não a forma (aparência) que eles serão exibidos. Nos parece sensato separar o conteúdo da apresentação (por questões de organização e padronização do código).

HELLO, WORLD! DJANGO

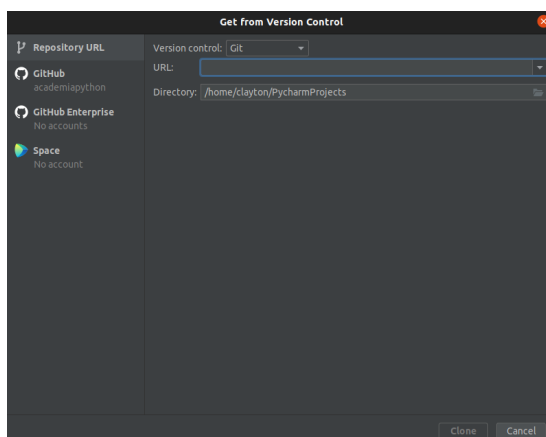
Criando o projeto

1. Abra o PyCharm e clique no botão Get from VCS localizado no canto superior da tela.



2. No campo URL, informe o seguinte endereço:

<https://github.com/academiapython/aplicacoes-web-responsivas-com-django-e-bootstrap.git>



3. Criar um branch a partir do commit inicial.
4. Instale o Django.

» python -m pip install django

5. Crie o projeto com o nome core.

» **django-admin startproject core**

6. Rode o projeto. Certifique-se que você está no diretório blog, pois é nele que o arquivo manage.py se encontra.

» **python manage.py runserver**

ENTENDENDO A ESTRUTURA DE UM PROJETO DJANGO

Antes de esmiuçarmos os arquivos criados pelo comando **startproject**, precisamos entender o que é um projeto Django. Para o Django, um projeto resume-se a um conjunto de arquivos de configuração que definem o comportamento global de nossas aplicações. Grosso modo, ele é equivalente ao web.xml dos projetos escritos em JSP e ao manifest.xml do Android.

Um projeto é composto de uma ou várias aplicações, e uma aplicação pode estar associada a vários projetos. Enquanto um projeto define o comportamento global da aplicação, um projeto contém um conjunto de funcionalidades bem definidas, como criação, alteração e remoção de usuários. Feita essa breve descrição, vamos entender a função de cada arquivo criado “automagicamente” pelo Django.

- **manage.py.** É o utilitário de linhas de comando que utilizamos para interagir com o Django.
- **_ _init_ _.py.** É um arquivo vazio que diz ao Python que o diretório onde ele se encontra deve ser tratado como um pacote.
- **urls.py.** Concentra as urls (rotas) de aplicações. Em breve entenderemos o fluxo de requisições django, ocasião em que a função desse arquivo ficará ainda mais clara.
- **asgi.py.** Informa que o projeto está utilizando ASGI, um padrão Python para desenvolvimento de aplicações e servidores web assíncronos.
- **wsgi.py.** A plataforma de deploy utilizada pelo Django é o WSGI, e esse configura nosso projeto para publicação em servidores WSGI.
- **settings.py.** É o arquivo que define as configurações do projeto. Esse arquivo possui as seguintes variáveis:
 - **BASE_DIR.** Diretório root do projeto.
 - **SECRET_KEY.** Chave criptográfica que será utilizada para assinar diversos elementos, tais como dados serializados, tokens de sessão, reset de passwords, mensagens, prevenção contra ataques do tipo cross-site, etc.
 - **DEBUG.** Quanto setada com valor True, apresenta os erros na tela de forma detalhada.

- **ALLOWED_HOSTS**. Define os hosts sobre os quais a aplicação ser executada.
- **INSTALLED_APPS**. Lista de aplicações instaladas em nosso projeto. Por padrão, o Django instala algumas aplicações utilitárias em nosso projeto, conforme você já deve ter percebido. É importante frisar que toda aplicação criada por nós precisa estar declarada no **INSTALLED_APPS**.
- **MIDDLEWARE**. Variável que registra todos os middlewares utilizados por nossa aplicação.
- **ROOT_URLCONF**. Indica qual é o arquivo de URL's de nosso projeto.
- **TEMPLATES**. Utilizamos esta variável para informar os diretórios em que se encontram os nossos templates.
- **WSGI_APPLICATION**. Informa o caminho do arquivo de configurações do WSGI.
- **DATABASES**. Declaração e configuração dos bancos de dados utilizados no projeto.
- **AUTH_PASSWORD_VALIDATORS**. Conjunto de utilitários para validação de senhas.
- **LANGUAGE_CODE**. Define o idioma do projeto.
- **TIME_ZONE**. Define o timezone do projeto.
- **USE_TZ**. Boolean que indica se o projeto utiliza timezone.
- **STATIC_URL**. Path no qual estão inseridos os arquivos estáticos do projeto (CSS, JavaScript, Imagens).
- **DEFAULT_AUTO_FIELD**. Indica que o Django irá criar chaves sequenciais para o projeto.

Criando a primeira aplicação Django.

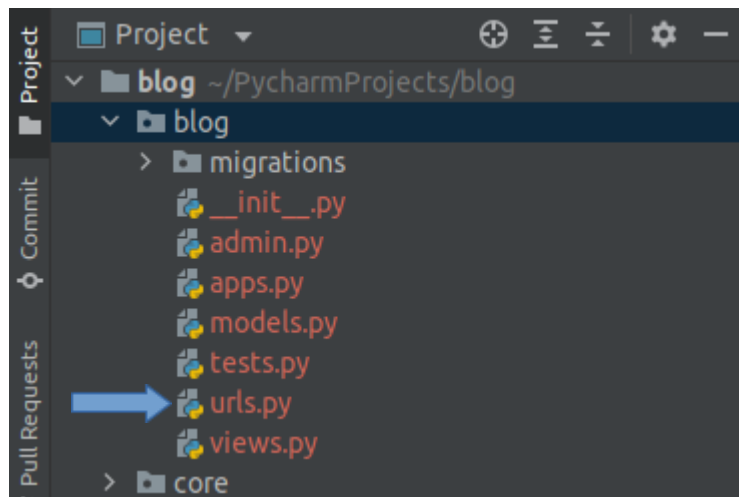
1. Crie o app blog.
» **django-admin startapp blog**
2. Registre o app blog na variável **INSTALLED_APP**, localizada no arquivo **/blog/settings.py**.

```
INSTALLED_APPS = [  
    ...  
    'django.contrib.staticfiles',  
    'blog.apps.BlogConfig',  
]
```

3. Crie, no arquivo `/blog/views.py`, um método que renderiza o conteúdo da página home.

```
from django.shortcuts import render  
from django.http import HttpResponse  
  
def home(request):  
    return HttpResponse("<h1>Hello, World!!!!</h1>")
```

4. Crie, na pasta do app `/blog`, um arquivo chamado `urls.py`.



5. Crie uma rota de acesso à home.

```
from django.urls import path  
from . import views  
  
urlpatterns = [  
    path('', views.home, name='home')  
]
```

6. Registre o arquivo **/blog/urls.py** no arquivo de URL's do projeto, localizado em **/core/urls.py**.

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

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

7. Rode o server.

» **python manage.py runserver**

8. Acesse a página inicial do projeto para visualizar o resultado.

Entendendo a estrutura de uma aplicação Django

Como você já sabe a diferença entre projeto e aplicações e tem pleno conhecimento da estrutura de um projeto, vamos analisar a estrutura de uma aplicação Django.

- **_ _init_ _.py**. É um arquivo vazio que diz ao Python que o diretório onde ele se encontra deve ser tratado como um pacote.
- **admin.py**. O módulo admin é uma ferramenta poderosíssima do Django. Entre as funcionalidades fornecidas por esse módulo estão o controle de acesso e execução de CRUD's sem que precisemos escrever uma única linha de código.
- **apps.py**. Define configurações que serão aplicadas apenas no app especificado.
- **models.py**. É o arquivo onde declaramos os models utilizados pela aplicação.
- **testes.py**. É o arquivo onde escrevemos os testes automatizados do app.
- **urls.py**. Arquivo em que registramos todas as urls da aplicação.
- **views.py**. Arquivo que contém as regras de negócio da aplicação.

Fluxo de requisições django – Parte I

Sem sombra de dúvidas, entender como o Django gerencia request/responses é suma importância para qualquer um que deseje dominar esse fantástico framework. Dada a

importância do assunto, iremos dedicar duas seções para ele. Por agora, iremos entender tudo o que fizemos até o momento.

Após a requisição passar por toda a pilha de protocolos TCP/IP, ela finalmente chegará ao servidor web onde nosso projeto se encontra hospedado. Este, por sua vez, irá encaminhar a requisição para nosso projeto Django, e é aqui onde se iniciam os trabalhos do framework.

A camada de middlewares é a porta de entrada do framework, sendo responsável por processar, de forma centralizada, todas as requisições que chegam ao nosso projeto Django. Esse processamento ocorrerá na ordem em que os middlewares aparecem na variável **MIDDLEWARE** localizada no arquivo **core/settings.py**, de cima para baixo. É importante frisar que nossos middlewares devem ser acrescentados ao fim de todos os middlewares declarados pelo Django.

Após isso, o URL Dispatcher irá entrar em ação. Ele receberá a requisição da camada de middlewares e irá entregá-la para a aplicação responsável pelo processamento. Para executar essa tarefa, a variável **ROOT_URLCONF** é de suma importância. Ela se encontra localizada no arquivo **settings.py** e sua finalidade é informar qual arquivo conterá todas as UR's do projeto, que em nosso caso é o **core/urls.py**.

A partir desse ponto, o URL Dispatcher irá buscar pela variável **urlpatterns** no arquivo **urls.py**, com intuito de localizar a aplicação(URL) que processará a requisição e devolverá o response apropriado.

É importante dizer que o fluxo de requisições Django não se encerra por aqui. Ainda há passos adicionais que serão detalhados posteriormente. Isso porque ainda não estudamos Templates, e essa explicação ficaria muito abstrata para aqueles que estão tendo o primeiro contato com o framework.

UM POUCO DE GIT FAZ BEM!

Nessa seção, iremos aprender o suficiente para utilizarmos Git em nossos projetos. Caso você já tenha um bom conhecimento de Git, sinta-se à vontade para pular essa seção ou consultá-la sempre que sentir necessidade. Conforme falado em aula, adiciono [aqui](#) o link para o vídeo em que Fábio Akita fala sobre sistemas de versões de uma maneira bem profunda.

O que é o Git?

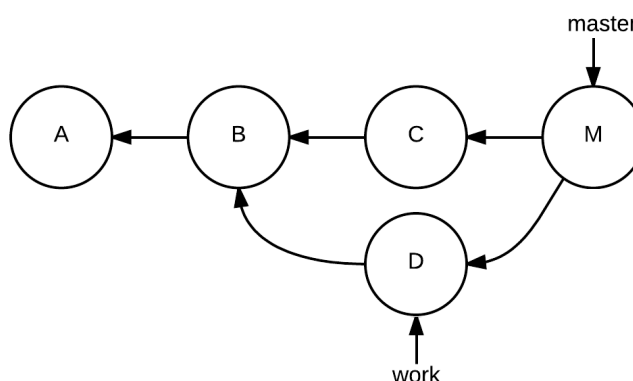
O Git é um Sistema de Controle de Versão Distribuído, criado por Linus Torvalds para o desenvolvimento do Kernel dos sistemas operacionais Linux. Ele se tornou praticamente uma unanimidade quando o assunto é controle de versões. Aqui vão alguns projetos famosos que utilizam Git: Debian, Android, Eclipse, Gimp, PHP, Reddit, VLC, Ruby on Rails e muitos outros.

Git Workflow

Todo projeto Git possui três fases diferentes: **working directory**, **staging area** e **git repository**. O **working directory** é o diretório que contém todos os arquivos que podem ser modificados. Depois de modificados, os arquivos são enviados para uma área de armazenamento temporária **staging area**. Quando os arquivos estiverem prontos, eles são salvos permanentemente no **git repository**. O git repository irá armazenar todo o histórico de commits.

Daremos uma olhada nos principais conceitos/elementos do Git:

- **Branch.** É uma “ramo/galho” nascido a partir do branch master. Ele carrega consigo um snapshot de todos os arquivos gerenciados pelo git repository.
- **Merge.** Ação de mesclar o conteúdo de um branch em outro. A figura abaixo ilustra o funcionamento dos branches e merges.



- **Gitignore.** Às vezes não queremos versionar determinados arquivos pelo fato de serem sensíveis (passwords) ou irrelevantes (arquivos de configuração do pycharm). Para resolver esse problema, utilizamos o arquivo `.gitignore`. Nele, adicionamos todos os arquivos que não desejamos versionar.

Principais comandos

- **Git init.** Cria um git repository em nossa estação de trabalho. Como nosso projeto foi criado no github, não precisamos executar esse comando em comando localmente.
» git init [nome do projeto]
- **Git clone.** Faz uma cópia completa do git repository em sua estação de trabalho. Esse repositório pode estar na nuvem ou localmente.

» **git clone /path/to/repository**

- **Git add.** Adiciona arquivos à **staging area**. Antes de ser definitivamente registrado no git repository, o arquivo precisa ter passado pela staging area.

» **git add [nome do arquivo]**

- **Git commit.** Move todos os arquivos da **staging area** para o **git repository**.

» **git commit -m "mensagem do commit aqui."**

- **Git push.** "Empurra" arquivos do repositório local para o branch nome_branch do repositório remoto.

» **git push origin nome_branch_remoto**

- **Git branch.** Comando utilizado para listar todas os branches do repositório ou remover um branch específico.

» **git branch** (lista todos os branches)

» **git branch -d nome_branch** (remove um branch específico)

- **Git checkout.** Pode ser utilizado para criar novos branches ou mudar de um branch para outro.

» **git checkout -d <nome_novo_branch>** (cria um novo branch)

» **git checkout <nome_branch>** (cria um novo branch)

- **Git merge.** Comando utilizado para mesclar as atualizações de um branch específico no branch que está ativo, ou seja, aquele que você está fazendo alterações.

» **git merge <nome_branch>**

- **Git pull.** Faz merge dos arquivos que estão no branch remoto com aqueles que estão em braches locais. Utilizado para obtermos alterações feitas por outros desenvolvedores.

» **git pull.**

PRIMEIRO MODEL

Criando o primeiro Model

1. Caso necessário, aprofunde o conceito de model consultando a [documentação oficial](#).
2. arquivo `/blog/models.py`, crie uma classe chamada Post.

```
from django.db import models
from datetime import datetime

class Post(models.Model):
    autor = models.CharField(max_length=255)
    titulo = models.CharField(max_length=255)
    subtitle = models.CharField(max_length=255)
    conteudo = models.TextField()
    data_publicacao = models.DateTimeField(default=datetime.now())
```

Configurando o Postgres

1. Abra o arquivo `core/settings.py`.
2. Substitua o conteúdo da variável **DATABASES** pelo conteúdo abaixo, conforme indicado na [documentação oficial](#) do Django.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'root',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

3. Instale o `psycopg2-binary`

» **pip install psycopg2-binary**

Caso você queira acessar a documentação oficial para obter informações sobre a escrita de dados no banco de dados Postgres em schemas diferentes do público, clique [aqui](#). Também há um artigo que detalha o processo de trabalhar simultaneamente com mais de um schema no postgres [aqui](#).

Migrations

1. Abra o terminal e execute o comando migrate.
» **python manage.py migrate**
2. Crie migrations para o app blog
» **python manage.py makemigrations blog**
3. Execute o comando migrate para o app blog.
» **python manage.py migrate**
4. Utilizando sua IDE de banco de dados favorita, observe a estrutura de tabelas criadas pelo Django.
5. Maiores detalhes sobre os comandos **migrate** e **makemigrations** podem ser consultados acessando a [documentação oficial](#) do Django ou revisitando a aula em que fizemos essas operações.

Introdução ao Django Admin

O Django-admin é uma ferramenta muito poderosa, e iremos utilizá-la para fazer executar algumas tarefas administrativas, como concessão de privilégios a usuários, criar o usuário administrador, entre outras tarefas. Além disso, iremos fazer cadastro, alteração, remoção e edição dos Posts de nosso blog. Para mais detalhes, acesse a [documentação oficial](#).

Crie o superusuário conforme indicado abaixo

1. Abra o terminal e crie o superusuário do projeto.
» **python manage.py createsuperuser**
2. Informe os dados requeridos.
3. Rode o servidor.

» **python manage.py runserver**

4. Abra o browser na URL <http://192.168.0.1/admin>
5. Digite as credenciais requeridas.
6. Registre o model Post no admin do Django. Faça isso abrindo o arquivo **blog/blog/admin.py** e adicionando o conteúdo abaixo:

```
from django.contrib import admin
from . import models

admin.site.register(models.Post)
```

7. Acesse novamente a aplicação admin e faça a criação de um Post.

CRIANDO NOSSO PRIMEIRO TEMPLATE

1. Crie, no diretório da aplicação **blog**, um diretório chamado **templates**.
2. Crie, dentro do diretório **blog/templates**, um diretório com o mesmo nome da aplicação, que no nosso caso deve se chamar **blog**.
3. Crie o arquivo **home.html** dentro do diretório **blog/templates/blog**. com o conteúdo indicado abaixo:

Seguem algumas considerações importantes a respeito dos templates:

- A. A variável **TEMPLATE**, presente no arquivo **settings.py**, descreve como o Django irá carregar e renderizar nossos templates. Por padrão, a variável **APP_DIRS** está setado como true. Dessa forma, o Django irá procurar este subdiretório em todos os apps declarados em **INSTALLED_APPS**.
- B. Nós poderíamos inserir nossos templates diretamente no diretório **templates**, ao invés de criar outro subdiretório chamado **blog**, mas isso seria uma péssima ideia em um cenário de reuso da aplicação em outros projetos. Caso existam mais de um template com o mesmo nome, o framework não saberá qual deles deverá ser carregado.
- C. Pelas razões apresentadas acima, devemos armazenar nossos templates seguindo as orientações apresentadas no início desta seção.

CONHECENDO UM POUCO DA API QuerySet

Quando criamos models, o Django nos disponibiliza automaticamente uma API que funciona como uma camada de abstração sobre a base de dados. Por meio dessa API, podemos executar diversas operações sobre os dados, tais como consultas, criação, alteração, remoção e atualização.

Nossos estudos serão direcionados para os problemas que precisamos resolver, dado que essa API é bastante extensa e não faz muito sentido empreender muitas horas nesse assunto. De início, veremos as funções essenciais, e no decorrer do curso iremos aprender a utilizar novas funções. Caso você se depare com problemas diferentes daqueles apresentados aqui, otimize seu tempo consultando a documentação oficial da [API QuerySet](#).

CRUD com a API QuerySet.

Criando objetos

1. Abra o shell.
» **python manage.py shell**
2. Importe a classe Post para o Interactive Console.
» **from blog.models import Post**
3. Instancie um objeto.
» **obj = Post(autor='clayton', titulo='interative', subtítulo='console',
conteúdo='este é o conteúdo')**
4. Utilize o método print() para visualizar os atributos do objeto.
» **print(obj.autor)**
5. Salve o objeto.
» **obj.save()**
6. Abra o DBeaver para confirmar a criação do registro na base de dados.

Recuperando objetos

Considerando que você esteja com o shell aberto e tenha importado o model Post

1. Recupere todos os objetos do model Post.
» **Post.objects.all()**
2. filtre objetos por um valor de chave primária (primary key / pk):
» **Post.objects.get(pk=3)**
3. Recupere os objetos pelo autor do Post.
» **Post.objects.filter(autor='clayton')**
4. Recupere os ativos cujos ID's sejam menores ou iguais a 4.
» **Post.objects.filter(id__lte=4)**
Obs.: less than or equal == lte
5. Recupere os ativos cujos ID's sejam menores que 4.
» **Post.objects.filter(id__lt=4)**
Obs.: less than == lt
6. Recupere os ativos cujos ID's sejam maiores ou iguais a 4.
» **Post.objects.filter(id__gte=4)**
Obs.: greater than or equal == gte
7. Recupere os ativos cujos ID's sejam maiores que 5.
» **Post.objects.filter(id__gt=5)**

Editando objetos

1. Atualize o título do post cujo ID = 3
» **obj = Post.objects.get(pk=2)**
» **obj.titulo = 'novo título'**
» **obj.save().objects.get(pk=2)**

Deletando objetos

1. Delete o objeto cujo id seja igual a 2.
» **obj = Post.objects.get(pk=2)**
» **obj.delete()**

2. Delete todos os objetos cujos ID's sejam maiores que 3.

» `maiores = Post.objects.filter(id__gt=3)`

» `maiores.delete()`

L

ISTANDO POSTS CADASTRADOS

1. Abra o arquivo **blog/view.py** e recupere todos os posts cadastrados no banco de dados.

```
from django.shortcuts import render
from .models import Post

def home(request):
    posts = Post.objects.all()
    context = {
        'posts': posts
    }
    return render(request, 'blog/home.html', context)
```

2. Edite o template **blog/index.html** para que ele carregue a lista de posts cadastrados em nosso banco de dados.

```
<ul>
    {% for post in posts %}
        <li>
            {{ post.titulo }}
        </li>
    {% endfor %}
</ul>
```


NAVEGANDO ENTRE PÁGINAS

1. Crie um arquivo chamado **post_detail.html** dentro do diretório **blog/templates**. Este template irá carregar todas as informações do post selecionado pelo usuário.
2. No arquivo **/blog/urls.py**, crie uma rota de acesso a um Post específico. Para isso, utilize o atributo ID que será passado por parâmetro pelo template **/blog/index.html**.

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

urlpatterns = [
    path('', views.home, name="home"),
    path('post/<int:post_id>/post_detail', views.get_post_detail, name="post_detail")]
```

3. Altere o arquivo **/blog/index.html**, incluindo nele um link para a página de detalhamento do post selecionado.

```
{% for post in posts %}
    <li>
        <a href="{% url 'post_detail' post.id %}">{{ post.title }} </a>
    </li>
{% endfor %}
```

4. Altere o conteúdo da página **/blog/post_detail.html** para exibir o conteúdo do post selecionado pelo usuário.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Conteúdo do post</title>
</head>
<body>
    <h2> {{post.titulo}} </h2>
    <h3> {{post.subtitulo}} </h3>
    <h3> {{post.autor}} </h3>
    <br />

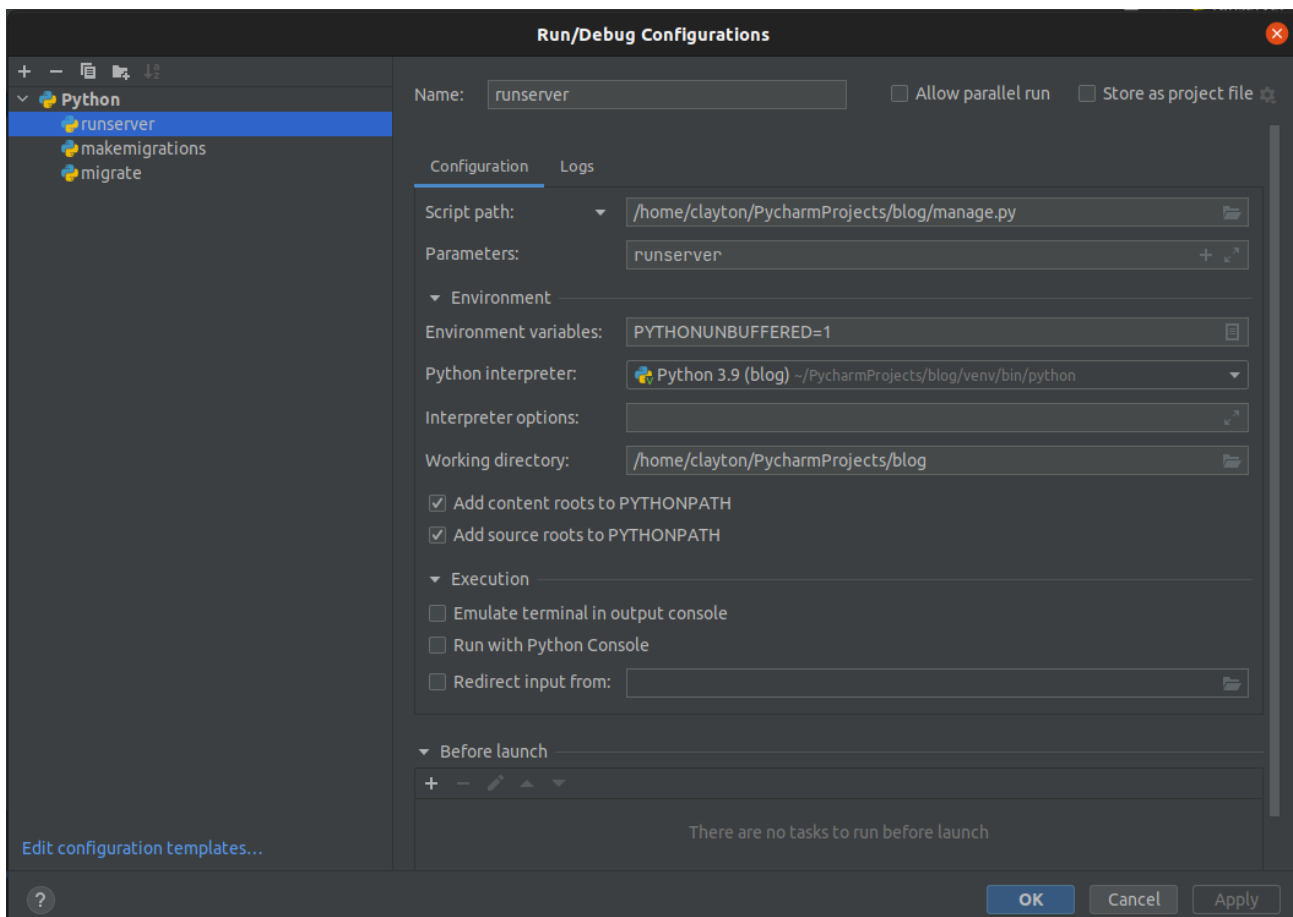
    <p> {{ post.content }} </p>

</body>
</html>
```

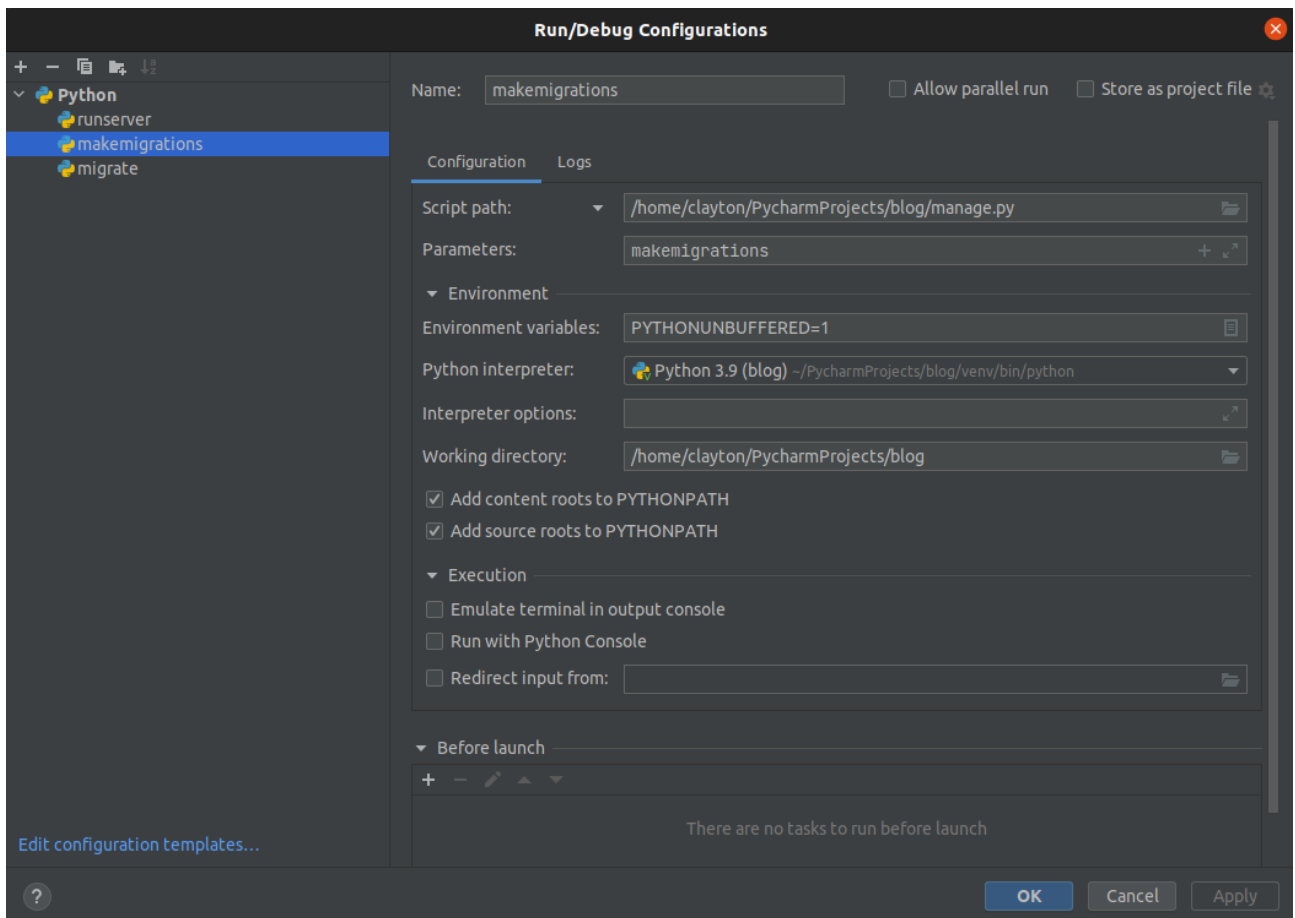
5. Rode o server, recarregue a página e veja a mágica acontecer!

CONFIGURANDO OS COMANDOS RUNSERVER, MAKEMIGRATIONS E MIGRATE NO PYCHARM

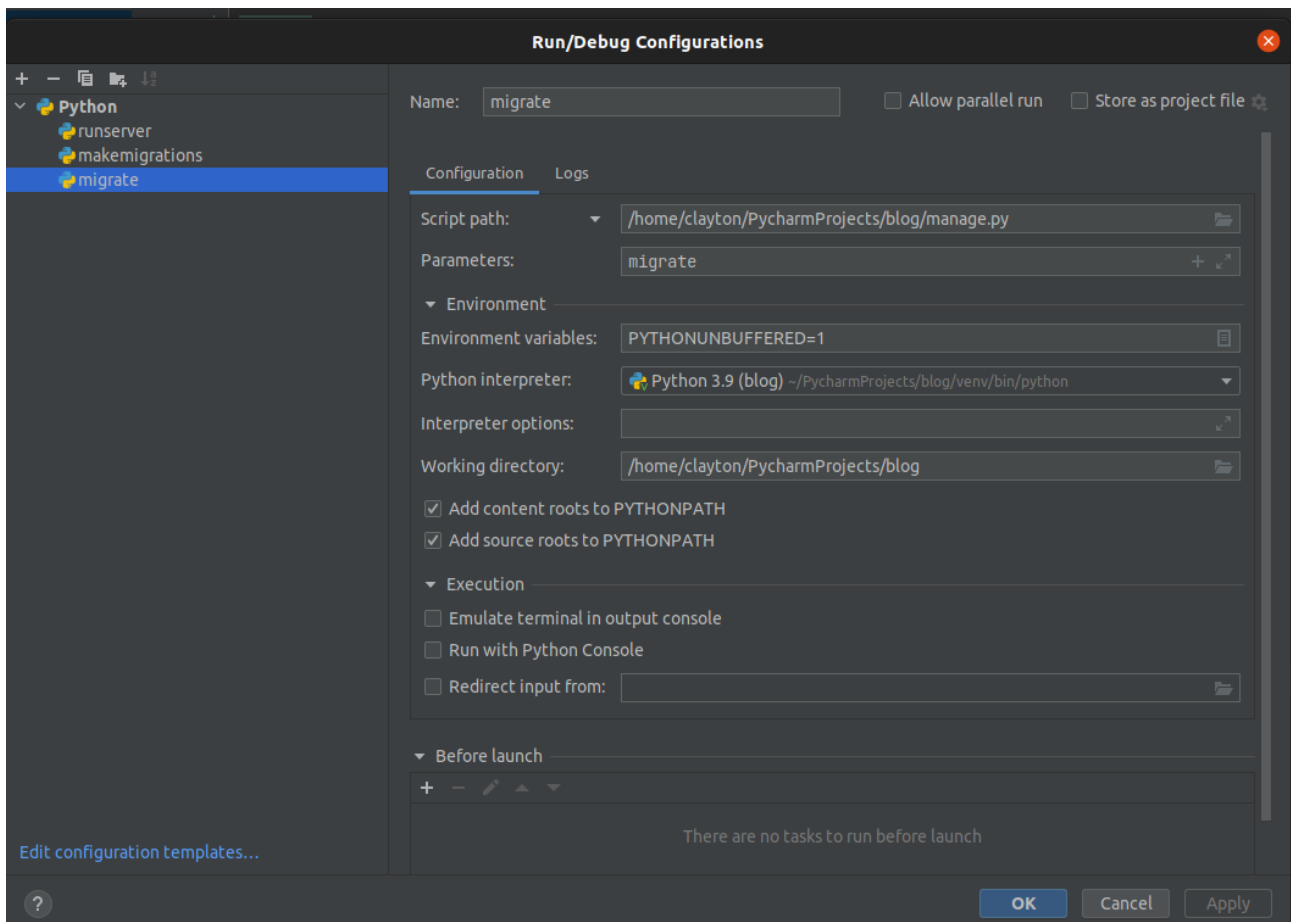
Configurações do comando python manage.py runserver



Configuração do comando python manage.py makemigrations



Configuração do comando python manage.py migrate



FLUXO DE REQUISIÇÕES – PARTE II

Iremos concluir nesta Seção o estudo das requisições no Django. Nosso estudo se iniciará após o URL Dispatcher identificar qual URL será responsável por processar a request. Uma vez obtida essa informação, ela será encaminhada para a camada de view.

Como já é de nosso conhecimento, a **camada de view** irá processar a lógica de negócio requerida, podendo ensinar a emissão de consultas à base de dados. Havendo necessidade de obter dados, a camada de view irá encaminhar suas solicitações à camada de **ORM**.

A camada de ORM é uma abstração sobre o banco de dados, proporcionando um grande número de operações de bancos de dados sem que tenhamos que escrever sentenças SQL, sejam elas DDL ou DML. Vimos a camada ORM entrar em ação quando estudamos a API QuerySet do Django. Por óbvio, temos uma camada adicional, que é a camada de dados. Ela

é representada por nosso banco de dados, que foi criado “automagicamente” pela camada de ORM quando executamos os comandos `makemigrations` e `migrate`.

De posse dos dados requisitados, a view irá inseri-los no contexto de um template. Este, por sua vez, carregará os dados em uma página HTML que será renderizada no browser do usuário.

DESMISTIFICANDO OS STATIC FILES

O Django trata arquivos .CSS, Javascript e imagens como “static files”. Por padrão, o Django provê a aplicação **`django.contrib.staticfiles`** para nos auxiliar no tratamento desses arquivos durante o desenvolvimento de nossas aplicações (**`DEBUG=True`**).

Quando executamos o comando `runserver`, a aplicação **`django.contrib.staticfiles`** irá servir ao browser todos os nossos arquivos estáticos. Entretanto, a equipe de desenvolvimento do Django nos orienta a não utilizar essa funcionalidade no ambiente de produção, dado que ela é ineficiente e insegura. Quando lançamos nosso código em produção, discutiremos estratégias para trabalhar com arquivos estáticos.

Antes de exemplificarmos o carregamento de um arquivo estático no template `home.html`, precisamos esclarecer a função de duas variáveis:

- **`STATIC_URL`**. Variável que define o caminho relativo para o diretório de arquivos estáticos da aplicação ou projeto.
- **`STATIC_ROOT`**. Variável que define o caminho absoluto (URL) para o local em que os arquivos estáticos serão armazenados. Essa variável indica o local em que o comando **`collectstatic`** irá inserir todos os arquivos estáticos de todas as aplicações.

A configuração dos arquivos estáticos deve seguir as seguintes etapas:

1. Nos assegurarmos que a aplicação **`django.contrib.staticfiles`** encontra-se instalada em nosso **`INSTALLED_APPS`**, localizada em **`core/settings.py`**.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    ...  
    'django.contrib.staticfiles',  
    'blog.apps.BlogConfig',  
]
```

2. Definir a variável **`STATIC_URL`** caso isso não tenha sido feito.

```
STATIC_URL = 'static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

3. Crie a estrutura de diretórios que armazenará os arquivos estáticos da aplicação seguindo o padrão **blog/static/blog**.
4. Utilizar o template tag abaixo para construir o caminho relativo de nossos arquivos estáticos. Em seguida, carregue a imagem em seu template utilizando a tag do html.

```
{% load static %}

```

5. Altere o arquivo `blog/urls.py` para possibilitar que a imagem seja carregada no ambiente de desenvolvimento.

```
urlpatterns = [
    path('', views.home, name='home'),
    .....
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

CONFIGURANDO EDITOR DE TEXTOS SIMILAR AO WORDPRESS (CKEDITOR)

1. Instale o CKEditor. Link da documentação oficial [aqui](#).
» `python -m pip install django-ckeditor`
2. Adicione `ckeditor` e `ckeditor_uploader` no `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ....
    'blog.apps.BlogConfig',
    'bootstrap5',
    'ckeditor',
    'ckeditor_uploader',
]
```

3. Insira a rota de acesso ao ckeditor no arquivo `core/urls.py`.

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

4. Abra o arquivo `core/settings.py` e informe ao Django qual o nome do diretório que armazenará os arquivos estáticos do projeto, caso não tenha feito ainda.

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

5. Configure o local em que o **CKEditor** irá salvar suas imagens. Para isso, abra o arquivo `core/settings.py` e adicione e crie a variável **CKEDITOR_UPLOAD_PATH**:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
CKEDITOR_UPLOAD_PATH = 'uploads/'

MEDIA_URL = 'images/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'static/images')
```

7. Carregue todas as dependências do **CKEditor** no diretório definido pela variável **STATIC_ROOT**.

» `python manage.py collectstatic`

8. Abra o arquivo `/blog/models.py`, importe o CKEditor e altere o tipo do campo conteúdo (`RichTextUploadingField`).

```
...
from ckeditor_uploader.fields import RichTextUploadingField

class Post(models.Model):
    ...
    conteudo = RichTextUploadingField(blank=False, null=False)
    ...

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

9. Abra o arquivo **blog/templates/home.html** e altere a forma como o Django irá interpretar o conteúdo. Nós queremos que ele renderize um RichTextField. Para isso, utilize a scape “ | safe ”.

```
...  
<p> {{ post.content | safe }} </p>  
...
```

10. Abra o Django Admin, cadastre um novo post e veja a mágica acontecer!!!

INTENSIVÃO BOOTSTRAP

O Bootstrap é o framework de desenvolvimento de aplicações web responsivas mais popular do mundo. E a partir de agora, iremos fazer um estudo intensivo e denso do framework Bootstrap. Para aqueles que assistiram Naruto, nosso treinamento será semelhante ao treino que nosso amado personagem fez para dominar o estilo vento. Não será fácil, mas uma vez dominado o conhecimento apresentado aqui, você alcançará um novo patamar na construção de seus layouts.

INSTALAÇÃO DO BOOTSTRAP

Formas de instalação. Existem diversas formas de instalação do Bootstrap em nosso projeto Django, mas a forma que iremos utilizar é via **CDN** (Content Delivery Network), seguindo o guia de instalação disponível na [documentação oficial](#) do Bootstrap. Outras formas de instalação estão disponíveis [aqui](#).

Seguem algumas vantagens do CDN:

- **Aprimorar os tempos de carregamento do site:** com a distribuição de conteúdo a partir de um servidor de CDN mais próximo dos visitantes do site (entre outras otimizações), os visitantes experimentam uma aceleração dos tempos de carregamento da página.

- **Reduzir os custos de largura de banda:** Por meio do armazenamento em cache e de outras otimizações, as CDNs conseguem reduzir a quantidade de dados que um servidor de origem precisa fornecer e, consequentemente, reduzem os custos de hospedagem para os proprietários de sites.
- **Aumentar a disponibilidade de conteúdo e a redundância:** grandes quantidades de tráfego ou falhas de hardware podem interromper o funcionamento normal do site. Graças à sua natureza distribuída, uma CDN consegue lidar com mais tráfego e resistir melhor às falhas de hardware do que muitos servidores de origem.
- **Aprimorar a segurança do site:** uma CDN pode aprimorar a segurança fornecendo mitigação de DDoS, aprimoramento dos certificados de segurança e outras otimizações.

Vamos criar nosso primeiro template seguindo, como sempre, a [documentação oficial](#).

Dinâmica:

Abrir o site, mostrar template, explicar as propriedades, copiar e colar o conteúdo para um template django.

CONTAINERS

No bootstrap existem três tipos de Containers: container default, container fluid e container breakpoint.

1. **.container.** Sempre ocupará uma proporção específica da tela. Entretanto, passará a ocupar toda a largura do dispositivo quando o tamanho da viewport for inferior a um “breakpoint” estabelecido pelo framework.
2. **.container-fluid.** Sempre ocupará a largura total da viewport.
3. **.container-{breakpoint}.** Ocupará 100% da largura do dispositivo quando as dimensões forem inferiores às dimensões do breakpoint especificado.

ENTENDENDO O CONCEITO DE MOBILE-FIRST

O bootstrap utiliza o conceito de mobile-first, ou seja, as páginas são primeiramente pensadas para os dispositivos móveis, e só após isso devemos pensar na exibição para as demais viewports. Isso ocorre porque a maior quantidade de acessos à internet é feita através desses dispositivos.

O bootstrap utiliza a CSS **@mediaquery** para identificar o tamanho da viewport, e com base nessa informação os componentes se ajustam à tela. Entretanto, esses ajustes são feitos tomando como base os **responsive breakpoints**.

[Responsives breakpoints](#) são tamanhos de telas pré definidos pelo bootstrap. São eles:

Breakpoint	Class infix	Dimensions
X-Small	<i>None</i>	<576px
Small	sm	≥576px
Medium	md	≥768px
Large	lg	≥992px
Extra large	xl	≥1200px
Extra extra large	xxl	≥1400px

MOBILE FIRST E CONTAINER BREAKPOINTS

Para entendermos o quanto de espaço nossos containers ocuparão na viewport, precisamos olhar a [tabela](#) abaixo:

	Extra small <576px	Small ≥576px	Medium ≥768px	Large ≥992px	X-Large ≥1200px	XX-Large ≥1400px
.container	100%	540px	720px	960px	1140px	1320px
.container-sm	100%	540px	720px	960px	1140px	1320px
.container-md	100%	100%	720px	960px	1140px	1320px
.container-lg	100%	100%	100%	960px	1140px	1320px
.container-xl	100%	100%	100%	100%	1140px	1320px
.container-xxl	100%	100%	100%	100%	100%	1320px
.container-fluid	100%	100%	100%	100%	100%	100%

Interpretando a tabela acima.

A classe **.container** ocupará 100% da largura da viewport quando a resolução desta for inferior a 576px. Ocupará 540px quando ela for maior que 576px. Ocupará, ainda, 720px quando a tela for maior ou igual a 768px e assim sucessivamente.

SISTEMA DE GRIDS

O bootstrap organiza seus layouts em um sistema de linhas, colunas e gutters. As linhas (.row) são separações horizontais, as colunas são separações verticais e os gutters definem o espaçamento entre colunas.

O bootstrap divide verticalmente a tela em 12 partes de tamanho igual, e as colunas podem assumir uma largura que varia entre 1 e 12 partes, desde que a soma total seja igual a 12. A especificação da largura é feita utilizando a classe ".col-[1-12]".

Antes de partirmos para a prática, é ter em mente as seguintes regras:

1. Todo conteúdo precisa estar empacotado dentro de um container.
2. Toda coluna precisa estar declarada dentro de uma linha. Se essa regra não for seguida, você terá problemas no alinhamento dos componentes.

NAVBAR

O componente [Navbar](#) normalmente fornece um conjunto de funcionalidades de navegação para a grande maioria dos sites que utilizam Bootstrap. Algumas coisas precisam ser ditas a respeito desse componente:

- 1) Por padrão, o navbar é responsivo.
- 2) Todo navbar precisa ser "envelopado" pela classes: **.navbar** e **.navbar-expand{-sm|-md|-lg|-xl|-xxl}** para habilitar o comportamento de collapsing.
- 3) O espaçamento e o posicionamento dos componentes são feitos através de [spacing](#) e [flex](#).
- 4) Por questões de usabilidade, o navbar precisa:
 - a) Ou ser "envelopado" pela tag html5 **<nav></nav>** ou possuir o atributo **role="navigation"**.
 - b) **aria-current="page"** e **aria-current="true"** para indicar aos leitores de tela se ele se encontra na página corrente ou não.
 - c) **aria-expanded={True | False}**. Informa aos leitores de tela o estado corrente do navbar.

A classe navbar suporta os seguintes subcomponentes:

- 1) **navbar-brand**. Brand significa marca. Utilizaremos esse elemento para exibir o logo ou nome do nosso site.
- 2) **.navbar-nav**. Utilizando essa classe, nosso navbar herda todos os poderes de um menu de navegação, incluindo suporte a menus dropdown.
- 3) **.navbar-toggler**. Botão utilizado para exibir/ocultar o **navbar-collapse**.
- 4) **.collapse** e **.navbar-collapse**. Componente que será exibido/ocultado por um **navbar-toggler**.

Por fim, vamos descrever sucintamente dois atributos importantes para implementação do comportamento responsivo de collapse do navbar. Ambos são adaptações do Bootstrap para os atributos html5 **data-toggle** e **data-target**. São eles:

- 1) **data-bs-toggle**. Cria uma espécie de “link” com elementos “colapsáveis”.
- 2) **data-bs-target**. Elemento que terá o comportamento de collapse por outro componente (toggle).

Com todas essas informações, já podemos entender 90% do funcionamento dos navbars. Inclusive, do navbar utilizado em nosso projeto. O restante será apresentado na aula.

FLEXBOX

A estrutura de layouts do Bootstrap é baseada no Flexbox Layout. Em linhas gerais, ele nos fornece uma forma “diferente” de posicionarmos nossos elementos. Em nosso curso, utilizaremos o Flexbox por meio das classes do Bootstrap.

A seguir, iremos resumir o comportamento das principais classes do Bootstrap que implementam o [Flexbox](#).

1. **d-flex**. O componente ocupará 100% da área disponível. Dessa forma, cada componente ocupa uma linha inteira. Em decorrência disso, havendo dois componentes dentro de uma classe **.d-flex**, eles serão adicionados um abaixo do outro.
2. **d-inline**. Os componentes ocuparão o espaço estritamente necessário. Por essa razão, eles serão posicionados um ao lado do outro enquanto houver espaço na linha.
3. **d-block**. Comportamento similar do **d-flex**. Os componentes ocuparão toda a largura disponível. Como consequência disso, haverá uma “quebra de linha” antes e depois de cada componente.
4. **d-none**. Quando utilizado em conjunto com breakpoints (ex: **d-lg-block**)), indica que o componente **não será exibido para o tamanho de viewport menores ou iguais ao informado**.

CARD

O componente [Card](#) é um componente muito utilizado em nosso projeto por sua flexibilidade. Ele pode conter seções de header, body e footer. Por ser uma extensão dos containers, aceitam todas as classes bootstrap e também possui algumas classes personalizadas que serão apresentadas na parte prática. As principais classes são as seguintes:

- **card-header.** Todo conteúdo “envelopado” por essa classe receberá formatação de um [header](#). Por óbvio, seu posicionamento será na parte superior do card.
- **card-title.** Classe utilizada para criação de títulos para nossos cards, como pode ser visto [aqui](#).
- **card-body.** Como você já deve ter imaginado, todo o conteúdo do card deverá ser “envelopado” por essa classe, conforme pode ser visto [aqui](#).
- **card-text.** Sempre que nossos cards possuírem texto, é recomendado que ele seja “envelopado” por esta [classe](#).
- **card-footer.** Sempre que desejarmos inserir um footer em nossos cards, é recomendado utilizar esta [classe](#).

GRID CARDS

[Grid Cards](#) possuem o mesmo princípio de funcionamento dos cards, exceto pelo fato de renderizar um ou mais cards em um sistema de grids. Os cards são renderizados da seguinte forma:

- `.row-cols-2`. Este código nos diz que teremos dois cards por linha.
- `.row-cols-md-4`. Este código nos indica que teremos quatro cards por linha desde que a coluna tenha tamanho maior ou igual ao definido pelo breakpoint “md”.

O funcionamento desse componente é bastante simples e será exemplificado durante a aula.

FORMULÁRIOS COM BOOTSRAP E DJANGO + ENCAMINHAMENTO DE EMAILS

Existem três formas de trabalhar com formulários no Django. Nesta seção, apresentaremos uma visão geral de cada uma delas, suas vantagens e desvantagens.

Formulários – Parte I. Criando e validando formulários com Bootstrap e Django.

Nesta seção, iremos apresentar um resumo de como podemos criar formulários HTML/Bootstrap, métodos de validação server side e manipulação dos estilos Bootstrap. Por essa abordagem, devemos fazer o seguinte:

1. Crie uma aplicação e um template, caso não tenha criado. Em caso de dúvidas, clique [aqui](#).
2. Crie, no template desejado, o código HTML do formulário. Aqui vão algumas informações importantes:
 - a. Todo formulário precisa de um método, que pode ser **GET (method="GET")** ou **POST (method="POST")**. Enquanto o primeiro é normalmente utilizado para recuperar informações do banco de dados, o segundo é indicado nas situações em que desejamos remeter dados para o servidor.
 - b. Todo formulário precisa de uma URL responsável pelo processamento das requisições. Isso é feito por meio do atributo **action="URL_AQUI"**.
 - c. Um botão do tipo **submit**.
 - d. Inclusão da tag `{% csrf_token %}` no formulário. Fazendo isso, protegemos nossos forms contra ataques de [Cross-Site Request](#).
3. Crie a view responsável por processar as requisições e informe-a no atributo action do formulário.
4. Crie uma rota de acesso à view no arquivo de urls.py da aplicação.
5. Receba os dados utilizando **request.Post.get(name_field)**.
6. Valide os dados recebidos.

Agora vem a pergunta que não quer calar ... como iremos informar ao usuário quais campos se encontram com erros de validação? Para fazer isso, precisamos dar uma olhadinha nas orientações do Bootstrap para implementação de validações [server side](#). Em resumo, temos o seguinte:

1. O Bootstrap utiliza as classes **.is-valid** e **.is-invalid** para indicar ao usuário que determinado campo está ou não válido.
2. A classe **.invalid-feedback** é utilizada pelo Bootstrap para apresentar ao usuário uma mensagem de erro.
3. A associação entre a mensagem de erro e o campo de input relacionado é dado pelo atributo **aria-describedby**. Este atributo deve possuir o id da divisão da classe **.invalid-feedback**.

A aplicação destes conceitos será apresentada em aula. Entretanto, você pode dar uma olhadinha na documentação referente às validações [server side](#), utilizar a sintaxe de interpolação do Django para manipular as classes do Bootstrap.

Formulários – Parte II. Criando e validando formulários com a classe Forms do Django.

Nesta seção, iremos criar e estilizar formulários utilizando a classe Form do Django.

Form – Primeiros passos

Da mesma forma que os models representam a estrutura de um objeto e seu comportamento, a [classe Form](#) descreve os campos de um formulário, seu comportamento e sua forma de apresentação nos templates (CSS e JavaScript). O Django utiliza a classe [Widget](#) para formatação HTML/CSS dos campos do formulário.

Renderizar um Form dentro de um template requer, com poucas alterações, os mesmos passos para renderizar um Model. Para isso, sigamos os seguintes passos:

1. Crie um arquivo **form_exemplo.py** na raiz de sua aplicação. Em nosso caso, o arquivo está localizado em **/exemplos/form_exemplo.py**.
2. Crie uma classe chamada **FormExemplo** que herda da classe **django.forms.Form**. A lista dos tipos de campos estão disponíveis [aqui](#). Maiores detalhes sobre a construção de formulários também podem ser vistos [aqui](#).

```
from django import forms

class FormExemplo(forms.Form):
    email = forms.EmailField(label="Email",
                             required=True,
                             max_length=60)
    password = forms.CharField(label="Senha",
                               widget=forms.PasswordInput())
```

3. Crie a view responsável pelo processamento das requisições. Lembre-se de importar a classe **FormExemplo**.

```
def processa_formulario_v2(request):
```

```

if request.method == 'POST':
    form = FormExemplo(request.POST)
    if form.is_valid():
        return HttpResponse('Formulário validado com sucesso!!!')
else:
    form = FormExemplo()
return render(request, '17_forms_parte_ii.html', {'form': form})

```

4. Crie um template que contenha um formulário. Utilizaremos o template criado na seção anterior. Entretanto, removemos o conteúdo html/bootstrap e inserimos o código abaixo.

```

<form method="POST" action="{% url 'processa_formulario_v2' %}">
    {% csrf_token %}
    {{ form }}
    <button type="submit">Cadastrar</button>
</form>

```

Note que estamos renderizando um objeto do tipo Form.

5. No arquivo **exemplos/urls.py**, crie uma rota de acesso ao template que contém o formulário, nomeando-a de **processa_formulario_v2**.
6. Acesse a url http://127.0.0.1:8000/exemplos/processa_formulario_v2 e veja a mágica acontecer!

Personalizado CSS do formulário

Agora que o objeto da classe Form está sendo renderizado em nosso template, chegou o momento de estilizá-lo com classes Bootstrap ou CSS. Esse trabalho é feito por meio de widgets, que nada mais são do que uma representação HTML dos atributos da classe form.

Espere um pouco... então qual seria diferença entre forms fields e widgets? Forms fields possuem a lógica de validação dos campos de nosso formulário, enquanto os widgets se ocupam com a renderização HTML dos campos. Vale registrar que os widgets precisam estar associados a um campo de formulário. Vejamos como isso será feito logo abaixo. Para mais detalhes, consulte a documentação neste [link](#).

1. No arquivo **/exemplos/forms_exemplo.py**, crie um atributo chamado mensagem conforme indicado abaixo:

```

mensagem = forms.CharField(widget=forms.Textarea(
    attrs={'class': 'form-control'}),
    label="Mensagem", min_length=10,

```



```
max_length=1000, required=True)
```

2. Salve o arquivo, recarregue a página e veja a mágica acontecer.

Observe que, embora nosso campo tenha validações de obrigatoriedade e tamanho, foi necessário criar um objeto do tipo widget e informar que ele deverá ser renderizado utilizando a classe [form-control](#) do Bootstrap.

Mas e se desejássemos utilizar mais de uma classe Bootstrap para esse campo? A resposta é simples! Tudo o que você precisa é inserir o nome das classes separado-os por vírgulas. Dessa forma, teríamos algo assim:

```
attrs={'class':'form-control, bg-primary'})
```

Agora, iremos alterar os demais campos do formulário para que utilizem a classe form-control do Bootstrap.

```
class FormExemplo(forms.Form):
    email = forms.EmailField(widget=forms.EmailInput(
        attrs={'class': 'form-control'}),
        label="Email", required=True,
        max_length=100, min_length=5)

    senha = forms.CharField(widget=forms.PasswordInput(
        attrs={'class': 'form-control'}),
        label="Senha", required=True,
        max_length=16, min_length=6)

    mensagem = forms.CharField(widget=forms.Textarea(
        attrs={'class': 'form-control'}),
        label="Mensagem", min_length=10,
        max_length=1000, required=True)
```

Formulários – Parte III. Criando formulários com a classe ModelForm

Nesta seção, criaremos nossa página de contato. Decidi criá-la utilizando a classe Model Forms do Django, ao invés de criar uma aplicação de exemplo. Não se preocupe porque todo o conteúdo visto até agora será aproveitado. Além dos formulários com Model Forms, estudaremos mecanismos de envio de email e inclusão de Captcha.

ModelForm – Primeiros passos

Iniciaremos essa seção falando sobre a principal diferença entre as classes Form e Model Forms, que é a seguinte: Todo Model Forms precisa estar associado a uma classe do tipo Model, e são os seus campos que serão carregados no template. As demais diferenças serão vistas no decorrer desta seção.

Essa classe é útil quando desejamos construir formulários a partir de entidades catalogadas em nosso banco de dados. Em nosso caso, todos os contatos serão armazenados no banco de dados.

Primeiramente, execute os seguintes passos:

1. Crie um model com os seguintes campos: assunto, primeiro nome, último nome, email e mensagem.

```
class FormContato(models.Model):
    assunto = models.CharField(choices=ASSUNTO_DROP_DOWN,
                              default="", max_length=100)
    primeiro_nome = models.CharField(max_length=100)
    ultimo_nome = models.CharField(max_length=100)
    email = models.EmailField(max_length=100)
    mensagem = models.TextField(max_length=1000)
```

2. Crie a variável ASSUNTO_DROP_DOWN no model do App contato.

```
ASSUNTO_DROP_DOWN = [
    ('', 'Selecione um assunto'),
    ('descontos', 'Descontos'),
    ('consultoria', 'Consultoria'),
    ('freelance', 'Freelance'),
    ('outros', 'Outros'),
]
```

3. Crie, na aplicação contato, um arquivo chamado **form_contato.py**. Este arquivo conterá os seguintes campos: First name, last name, email e mensagem.

```
from django.forms import ModelForm
from contato.models import FormContato

class FormContato(ModelForm):
```

```
class Meta:
    model = FormContato
    fields = '__all__'
```

- Configure o arquivo de urls da aplicação

```
urlpatterns = [
    path('', views.contato, name='contato'),
    path('mensagem', views.processa_contato, name='mensagem'),
]
```

- Renderize o formulário no arquivo **templates/contato.html**.

```
.....

<div class="col-xl-5 p-5">
    <form method="post" action="{% url 'mensagem' %}">
        {% csrf_token %}
        <div class="card p-5">
            {{ form }}
            <div class="card-footer ps-0 pt-3 bg-white">
                <button type="submit" class="btn
                    btn-primary">Cadastrar
            </button>
        </div>
    </div>
</form>
</div>

.....
```

- Volte ao arquivo **contato/form_contato.py** e configure a renderização dos campos inserindo a classe **form-control** do Bootstrap.

```
....
widgets = {
    'assunto': forms.Select(attrs={'class': 'form-control -select mb-3'}),
    'primeiro_nome': forms.TextInput(attrs={'class': 'form-control -3'}),
    'ultimo_nome': forms.TextInput(attrs={'class': 'form-control mb-3'}),
    'email': forms.EmailInput(attrs={'class': 'form-control mb-3'}),
    'mensagem': forms.Textarea(attrs={'class': 'form-control mb-3'}),
```

```
}  
....
```

7. Reinicie o servidor e veja se o formulário foi carregado corretamente.

As configurações mais detalhadas serão apresentadas durante a aula.

Sobre o encaminhamento de emails

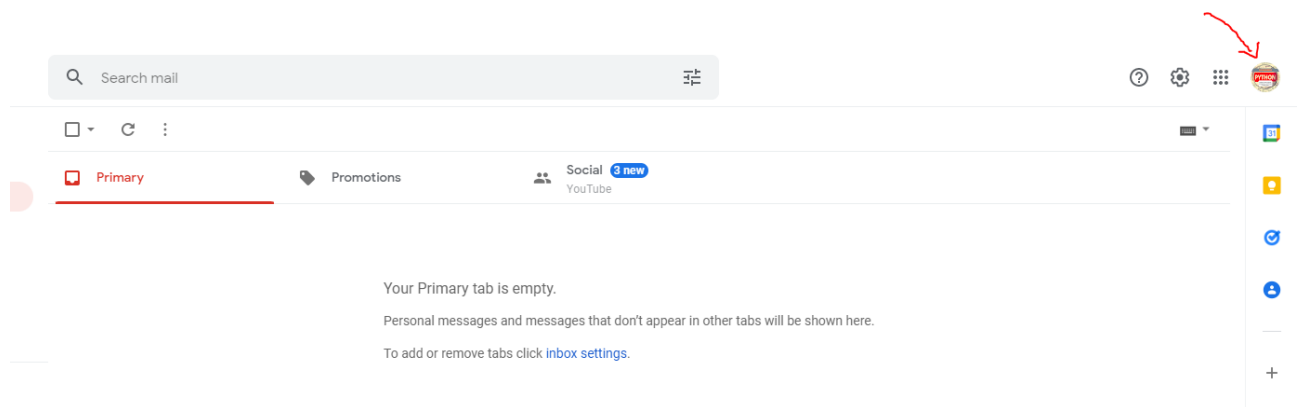
A grande maioria dos sistemas precisa encaminhar e-mails, e o Django, como era de se imaginar, provê uma interface simples e robusta para encaminhamento de emails. Todo esse trabalho é feito por meio da lib [smtplib](#).

Nesta seção, faremos encaminhamento de e-mails utilizando a smtplib, prevenção de header injection e strip_tags e personalizaremos o template html da mensagem de email.

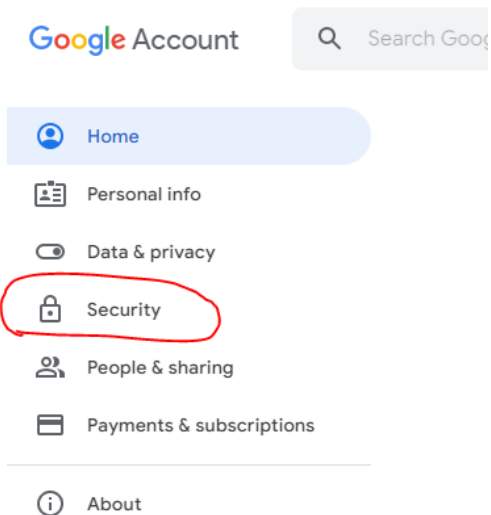
Configurando servidor SMTP do Gmail

Primeiramente, configure o gmail conforme indicado abaixo:

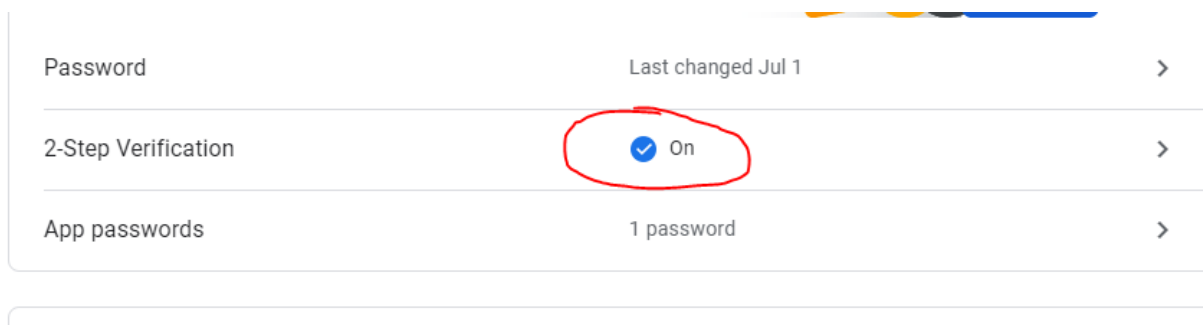
1. Clique na imagem do seu perfil e, após isso, clique em Gerenciar seu perfil (Manage your google account).



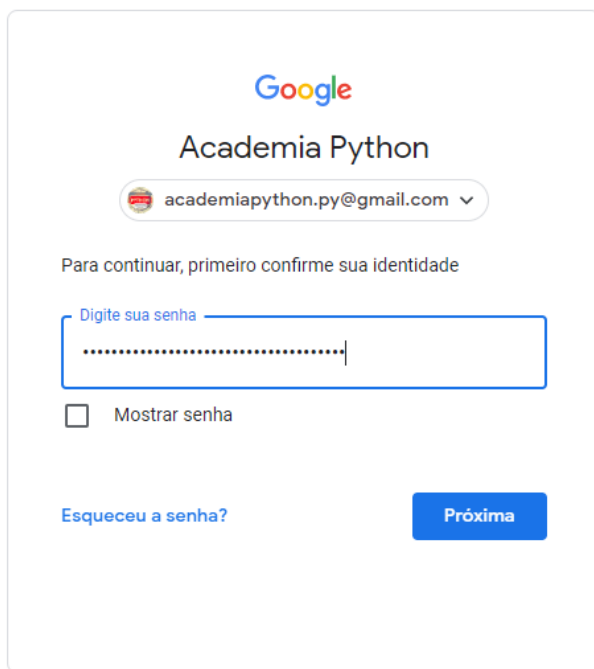
2. Clique na opção segurança (security)



3. Marque a opção Step verification (Verificação em duas etapas)



4. Após isso, clique na opção App passwords (Senha de app). Após isso, o Gmail solicitará que você informe a senha de seu email.



Google

Academia Python

academiapython.py@gmail.com ▼

Para continuar, primeiro confirme sua identidade

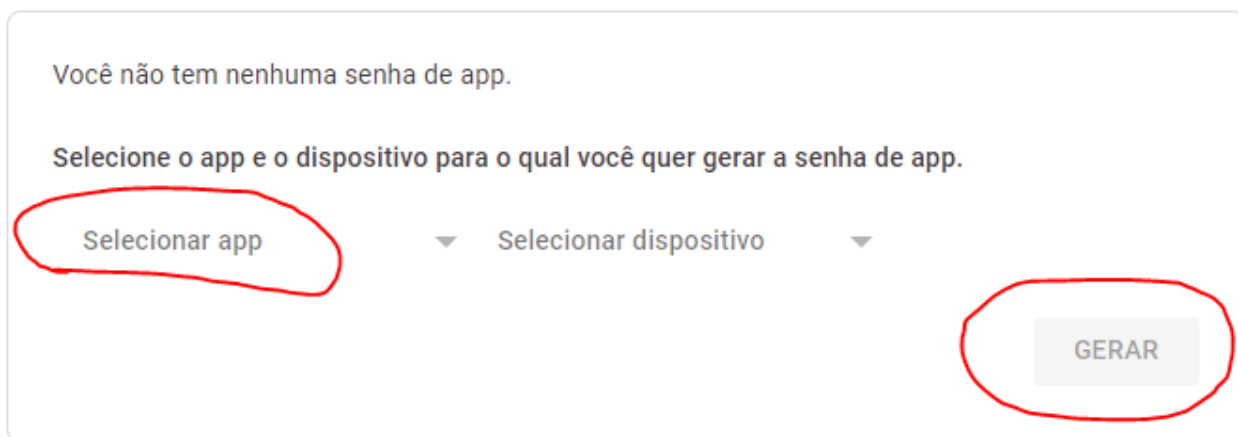
Digite sua senha

.....|

☐ Mostrar senha

[Esqueceu a senha?](#) [Próxima](#)

5. Clique na opção selecionar app. Após isso, escolha a opção Outro.



Você não tem nenhuma senha de app.

Selecione o app e o dispositivo para o qual você quer gerar a senha de app.

Selecionar app ▼ Selecionar dispositivo ▼

GERAR

6. Digite o nome do app e clique em gerar.

Senhas de app permitem que você faça login na sua Conta do Google a partir de apps em dispositivos que não sejam compatíveis com a verificação em duas etapas. Como só será necessário informar a senha uma vez, você não precisa memorizá-la. [Saiba mais](#)

Você não tem nenhuma senha de app.

Selecione o app e o dispositivo para o qual você quer gerar a senha de app.

send_email X

GERAR

7. Copie o valor da senha de 16 dígitos porque ela será utilizada nas configurações do Django.

Sua senha de app para seu dispositivo

cl5s zey lt xkq

Como usar

Acesse as configurações da sua Conta do Google no aplicativo ou dispositivo que você está tentando configurar. Substitua sua senha pela senha de 16 caracteres mostrada acima. Assim como sua senha normal, esta senha de app concede acesso total à sua Conta do Google. Não é necessário memorizá-la, por isso não a anote ou a compartilhe com outras pessoas.

CONCLUÍDO

Configurando o settings.py para encaminhamento de emails utilizando STMP do Gmail Para que os emails sejam encaminhados pelo Django, precisamos acrescentar algumas informações de configuração no arquivo **settings.py**.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'Digitar aqui o email remetente'
EMAIL_HOST_PASSWORD = 'Inserir aqui a senha'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
DEFAULT_FROM_EMAIL = 'default from email'
```

Feitas essas configurações iniciais, já podemos realizar o envio de email utilizando o Django. Esse envio requer os seguintes campos sejam informados:

1. **subject.** Assunto do email (string).
2. **message.** Conteúdo do email (string)
3. **from_email.** Remetente (string). Se nada for informado, o Django pegará o valor contido na variável `DEFAULT_FROM_EMAIL`.
4. **recipient_list.** Lista de emails dos destinatários (strings).
5. **fail_silently.** Variável booleana que indica se desejamos que exceções sejam levantadas na eventual ocorrência de erros. O valor **fail_silently=False** indica que a exceção `smtpplib.SMTPException` caso tenhamos erros no encaminhamento de emails.

Utilizaremos os dados informados pelo usuário para alimentar os campos requeridos pelo método `send_email()`, conforme descrito abaixo. Esse método será chamado pelo método `processa_contato()` logo após a validação dos dados submetidos pelo usuário.

```
def enviar_email(contato):
    send_mail(contato.cleaned_data['assunto'],
              contato.cleaned_data['mensagem'],
              settings.EMAIL_HOST_USER,
              [contato.cleaned_data['email']],
              fail_silently=False)
```

IMPLEMENTANDO SISTEMA DE AUTENTICAÇÃO CONTROLE DE ACESSO (EM CONSTRUÇÃO)

A grande maioria dos sistemas precisa de mecanismos de autenticação eficientes, robustos e seguros. Para nossa alegria, o Django nos fornece tudo o que precisamos para autenticar nossos usuários.

Antes de prosseguirmos, gostaria de consignar que alguns detalhes serão propositadamente omitidos para que você faça as configurações necessárias. Afinal de contas, você já é um adulto no django a essa altura do curso.

Realizando cadastro de usuários

Basicamente, esse sistema é composto pelos seguintes elementos: **usuários**, **grupos** e **permissões**. Enquanto os usuários representam pessoas que interagem com o sistema, as permissões nos dizem o que determinado usuário pode fazer e os grupos nos auxiliam na criação de perfis de acesso que aglutinam um conjunto de permissões.

Feita essa breve introdução, iremos criar uma aplicação para gerenciar as contas de usuários. Após isso, implementaremos métodos de autenticação e, por fim, mecanismos de autorização.

O model [User](#) é o coração do sistema de autenticação do Django. Em princípio, utilizaremos essa classe para fazermos cadastros no sistema. Para realizar cadastros, execute os seguintes passos:

1. Instale o widget-tweaks. Não preciso mais citar, a essa altura do campeonato, que precisa configurar essa lib e carregá-la no template apropriado.
» **`pip install django-widget-tweaks`**
2. Após ter criado a aplicação contas, crie um arquivo de urls para esta aplicação. Nele, insira uma rota de acesso à view **criar_conta**.

```
urlpatterns = [
    path('', views.criar_conta, name='criar_conta'),
]
```

3. Na aplicação contas, crie um arquivo chamado **usuario_form.py**. Observe que a classe User será o model de nosso form.

```
class Meta:
    model = User
    fields = ['first_name', 'last_name', 'username', 'email', 'password']
    labels = {
        'username': 'Username'
    }

    widgets = {
        'first_name': forms.TextInput(attrs={'class': 'form-control'}),
```

```

        'last_name': forms.TextInput(attrs={'class': 'form-control'}),
        'username': forms.TextInput(attrs={'class': 'form-control'}),
        'email': forms.EmailInput(attrs={'class': 'form-control'}),
        'password': forms.PasswordInput(attrs={'class': 'form-control'}),
    }

```

4. Ainda no arquivo **usuario_form.py**, transforme todos os campos do model como obrigatórios, já que apenas o campo username atende esse requisito. Lembre-se de inserir esse código antes da **classe Meta**.

```

def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)
    self.fields['first_name'].required=True
    self.fields['last_name'].required = True
    self.fields['username'].required = True
    self.fields['email'].required = True
    self.fields['password'].required = True

```

5. No arquivo **contas/views.py**, capture os dados submetidos via POST.

```

def criar_conta(request):
    if request.method == 'POST':
        profile = PerfilForm(request.POST)
        if profile.is_valid():

            usr = User.objects.create_user(
                first_name=profile.cleaned_data['first_name'],
                last_name=profile.cleaned_data['last_name'],
                username=profile.cleaned_data['username'],
                email=profile.cleaned_data['email'],
                password=profile.cleaned_data['password']
            )

            usr.save()
            return redirect('login')

        else:
            return render(request, 'contas/criar_conta.html', {'form': profile})
    else:
        return render(request, 'contas/criar_conta.html', {'form':
            PerfilForm()})

```

6. Crie, conforme apresentado durante as aulas, dentro da aplicação contas, o template **criar_conta.html**. A essa altura do campeonato, você já sabe onde e como inserir templates!

Com esse código, você já terá o cadastro de usuários funcionando perfeitamente!!!

Validando preenchimento dos campos com HTMX

Conforme podemos observar na [documentação oficial](#), o HTMX é uma biblioteca capaz de manipular, sem o uso de JavaScript, a estrutura do Document Object Model (DOM). Durante esse curso, iremos utilizar os aspectos gerais dessa tecnologia. Posteriormente, lançarei um curso completo sobre o assunto. Para implementar as validações, siga os passos abaixo:

1. Insira a dependência CDN no header do template.

```
<script src="https://unpkg.com/htmx.org@1.8.0"
integrity="sha384-cZuAZ+ZbwnNRnrKi05G/fjBX+azI9DN0kNYysZ0I/X5ZFgsmMiBXg
DZof30F5ofc" crossorigin="anonymous"></script>
```

2. Abra o arquivo de contas/urls.py e insira rotas para processamento das validações.

```
htmx_urlpatterns = [
    path('criar_conta/htmx_valida_username', views.htmx_valida_username,
name='htmx_valida_username'),
    path('criar_conta/htmx_valida_senha', views.htmx_valida_senha,
name='htmx_valida_senha'),
    path('criar_conta/htmx_valida_email', views.htmx_valida_email,
name='htmx_valida_email'),
]

urlpatterns += htmx_urlpatterns
```

3. Crie um template chamado **contas/feedback_form_validation.html**. Esse template será “empurrado” assíncronamente para o front-end.

```
<div id="usr-val" hx-swap-oob="true">
    <label style='color:{{cor}};'> {{ error_usrname }} </label>
</div>

<div id="usr_email" hx-swap-oob="true">
    <label style='color:{{cor}};'> {{ usr_email }} </label>
</div>

<div id="error_pwd" hx-swap-oob="true">
    <label style='color:red;'> {{ error_pwd }} </label>
</div>
```

```
<div id="botao_submit" hx-swap-oob="true">
  <button class="w-100 btn btn-primary btn-lg" {{ st_submit }} type="submit">
Criar conta
  </button>
</div>
```

4. Crie o método para validar o username.

```
def htmx_valida_username(request):

    context = {'error_username': 'Username indisponível', 'st_submit':
'disabled', 'cor': 'red'}

    usernameParam = request.POST.get('username')

    if not User.objects.filter(username=usernameParam):
        context['error_username'] = 'Username disponível'
        context['cor'] = 'green'

    if PerfilForm(request.POST).is_valid():
        context['st_submit'] = ''

    str_template = render_to_string('contas/feedback_form_validation.html',
context)
    return HttpResponse(str_template)
```

Observe que o conteúdo do template será preenchido de acordo com as validações especificadas no código acima.

5. Crie um método para validar se as senhas coincidem.

```
def htmx_valida_senha(request):

    context = {'error_pwd': 'As senhas não coincidem', 'st_submit': 'disabled',
'cor': 'red'}
    pwd_confirm = request.POST.get('pwd_confirm')
    password = request.POST.get('password')

    if pwd_confirm == password and PerfilForm(request.POST).is_valid():
        context['error_pwd'] = ''
        context['st_submit'] = ''

    str_template = render_to_string('contas/feedback_form_validation.html',
context)
    return HttpResponse(str_template)
```

6. Crie um método para validar o email.

```
def htmx_valida_email(request):
    context = {'usr_email': '', 'st_submit': 'disabled', 'cor': 'red'}
    email = request.POST.get('email')

    if not validou_email(email):
        context['usr_email'] = 'Email inválido.'
    if User.objects.filter(email=email):
        context['usr_email'] = 'Email já se encontra cadastrado.'
    if PerfilForm(request.POST).is_valid():
        context['usr_email'] = ''
        context['st_submit'] = ''

    str_template =
    render_to_string('contas/feedback_form_validation.html', context)
    return HttpResponse(str_template)
```

Maiores informações serão dadas durante as aulas.

Login com Redes Sociais

Iremos apresentar de forma sucinta e objetiva o passo-a-passo para configuração dos logins com redes sociais utilizando a famosa biblioteca [allauth](#).

1. Instale e configure a lib allauth.
» **pip install django-allauth**
2. Os apps requeridos para autenticação com allauth.

```
'allauth',
'allauth.account',
'allauth.socialaccount',
'allauth.socialaccount.providers.google',
'allauth.socialaccount.providers.github',
'allauth.socialaccount.providers.facebook',
```

3. Verifique se o app **django.contrib.sites** está no **INSTALLED_APPS**.
4. Verifique se o **django.template.context_processors.request** está no seu arquivo **settings.py**.

```
TEMPLATES = [
    {
```

```

        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.request',
            ],
        },
    ],
]

```

5. Adicione o código abaixo no arquivo **settings.py**.

```

AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
]

```

6. Ainda no arquivo **settings.py**, adicione o código abaixo.

```

SITE_ID = 1

```

7. Adicione as urls do allauth no arquivo **core/urls.py**.

```

path('accounts/', include('allauth.urls')),

```

8. Rode as migrations relativas ao allauth.

```

python manage.py migrate

```

9. Acesse <http://127.0.0.1:8000/accounts/> e observe todas as urls relacionadas ao allauth. Experimente as funcionalidades fornecidas pelo allauth.

Ainda será necessário fazer configurações no django-admin, bem como criação de contas de desenvolvimento no Google, Facebook e Github. Isso será apresentado durante a aula em vídeo.

EXTENDS USER MODEL, MANAGERS E SIGNALS

Desafio de programação (Projeto base da seção)

Nesta seção, faremos extends da classe User Model, personalizaremos nosso próprio manager e também estudaremos sobre signals.

Nesse sentido, eu gostaria de te desafiar a criar um projeto com os seguintes requisitos **obrigatórios**:

1. Crie um projeto chamado manager_usrmodel_signal.
2. Crie uma aplicação chamada app_manager.
3. Faça todas as configurações necessárias para essa aplicação.
4. Crie um model chamado Perfil que possua os seguintes atributos:
 - a. data_nascimento (Obrigatório)
 - b. foto (Opcional)
5. Execute as migrations. Certamente você irá se deparar com um erro após tentar rodar as migrations. Observe o log e resolva o problema!
6. Onde as imagens serão salvas? Faça essa definição.
7. Disponibilize o model para edição via django-admin.
8. Crie um superusuário.
9. Cadastre alguns usuários no banco de dados.
10. No Django-Admin, você consegue visualizar as imagens no browser quando clica sobre o campo foto? Caso não consiga, lembre-se de configurar a variável MEDIA_URL e MEDIA_ROOT.

Caso você queira um desafio um pouco maior, implemente os seguintes requisitos **opcionais**:

11. Crie um template para listar perfis cadastrados.
12. Crie cadastrar perfis. Utilize um model form para fazer isso.
13. Configure o arquivo de urls do projeto para apresentação de imagens e outros arquivos estáticos.

Managers

Conforme definido na documentação oficial do Django, um Manager é a interface através da qual as operações de consulta ao banco de dados são fornecidas a todos os Models do Django. Por padrão, todo manager fornecido pelo Django tem o manager chamado objects. Talvez sua mente esteja gritando: “EUREKA!”, agora eu sei o que significa Person.objects.

Aqui vai uma informação importante. Só devemos personalizar managers em duas situações:

1. Adicionar métodos adicionais aos managers existentes

2. Modificar o comportamento de recuperação dos dados (função all())

Um cuidado adicional deve ser tomado quando estivermos trabalhando com mais de um manager. O primeiro manager declarado será considerado como padrão pelo Django.

Alterando o nome do manager

```
class Perfil(models.Model):
    data_nascimento = models.DateField(null=False)
    foto = models.ImageField(null=True, blank=True, upload_to='app_manager/imgs/')
    manager = models.Manager()
```

Criando Custom Manager

```
class PerfilManager(models.Manager):
    def tot_img_nulas(self):
        return self.filter(foto__exact='').count()
```

Utilizando Custom Manager

```
class Perfil(models.Model):
    data_nascimento = models.DateField(null=False)
    foto = models.ImageField(null=True, blank=True, upload_to='app_manager/imgs/')
    manager = models.Manager()
    imgcounter = PerfilManager()
```

Utilizando manager renomeado

```
Perfil.manager.all()
```

Utilizando manager personalizado

```
Perfil.imgcounter.tot_img_nulas()
```

Alterando o comportamento do método all()

```
class PerfisComFoto(models.Manager):
    def get_queryset(self):
        return super().get_queryset().exclude(foto='')
```



```
class Perfil(models.Model):
    data_nascimento = models.DateField(null=False)
    foto = models.ImageField(null=True, blank=True, upload_to='app_manager/imgs/')
    manager = models.Manager()
    perfisComFoto = PerfisComFoto()
```

Extends User Model

Existem quatro formas fazer extends da classe User Model, e cada uma delas possui vantagens e desvantagens associadas. A seguir, apresentarei um resumo de cada abordagem.

Proxy Model

Esta abordagem é indicada quando você deseja estender a classe User Model e não precisar criar novas tabelas ou armazenar informações adicionais no banco de dados, mas simplesmente adicionar métodos ou alterar a forma como as informações são recuperadas no banco de dados via Manager.

