O que esse curso irá abordar?

1. **Linux**
   - 1.1. Conceitos básicos
   - 1.2. Estrutura de arquivos
   - 1.3. Comandos básicos

2. **Ambientes Virtuais do Python**
   - 2.1. Criar ambientes virtuais
   - 2.2. Explicação do que são os ambientes virtuais e suas utilidades
   - 2.3. Como acessar e desativar ambientes virtuais criados
   - 2.4. O que é e como funciona o gerenciador de pacotes do Python (PIP)

3. **Introdução ao Django**
   - 3.1. Criar uma estrutura básica para fazer uma introdução ao Django
   - 3.2. Explicar sobre a estrutura MVT do Django e fazer uma comparação com MVC (Citando o .NET / C# como exemplo).
   - 3.3. Conceitos básicos

4. **Desenvolvimento Web com Django**
   - 4.1. Visão Geral do Desenvolvimento Web com Django
     - 4.1.1. Modularização
     - 4.1.2. Segurança
     - 4.1.3. Variáveis de Ambiente
     - 4.1.4. Conceitos de CI/CD
     - 4.1.5. Conceitos básicos do Git
     - 4.1.6. Configuração do VSCode
     - 4.1.7. Criar aplicações
     - 4.1.8. Versionamento de código com o GitHub
     - 4.1.9. Modelo de dados
     - 4.1.10. Migração do modelo para banco de dados
     - 4.1.11. Listando o modelo de dados no painel administrativo do Django
     - 4.1.12. Implementação de testes do modelo de dados
     - 4.1.13. Branch Merge
   - 4.2.

Ambientes virtuais Python

- Criar ambiente virtual
    - python3 -m venv *nome*

- Entrar no ambiente virtual
    - . /path/to/venv/bin/activate

- Sair do ambiente virtual
    - deactivate

Com o ambiente virtual ativo:

- Instalar o django: pip install Django
    - https://pypi.org/project/Django/

- Iniciar o projeto do blog no diretório desejado:
    - django-admin startproject *nome* .

Observações:
Agora que o projeto foi criado, não é mais necessário utilizar o ***django-admin***. Será utilizado o arquivo *manage.py*.

Iniciar a aplicação para ver se está tudo funcionando normalmente.

Modularização:

É necessário este conceito, pois deixará o código mais organizado, e podemos configurar facilmente tanto o ambiente de produção como local (de desenvolvimento).

A primeira configuração será modularizar as configurações, mais especificamente o arquivo *settings.py*.
Dentro do nosso projeto principal, vamos criar uma nova pasta chamada *settings*. E dentro desta pasta devemos criar um arquivo chamado *__init__.py* pois queremos acessar essa nova pasta como se fosse um módulo, e então moveremos o arquivo *settings.py* para dentro dessa nova pasta e renomearemos o arquivo para *base.py*.

Se tentarmos inicializar a nossa aplicação, veremos um erro, pois o Django não consegue mais encontrar as 'configurações' ou mais especificamente, o arquivo *settings.py* que movemos para a pasta *settings* e renomeamos para *base.py*. Isso ocorre porque devemos especificar no arquivo *manage.py* o caminho para que o Django possa saber onde está este arquivo de configuração

Veja no exemplo abaixo:

```
def main():
    """Run administrative tasks."""
    os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'ByteBuilders.settings')
```

Na linha os.environ temos no final *'ByteBuilders.settings'*, essa é a informação que precisamos atualizar *'ByteBuilders.settings.base'*, agora estamos dizendo ao Django para procurar as configurações dentro da pasta *settings* recém criada.

1. **Local vs Production**

Lembra que mudamos o nosso arquivo *settings.py* para *base.py*? Pois bem, essa alteração foi para que possamos ter configurações para o nosso ambiente local e para o ambiente de produção. Então para darmos seguimento, devemos criar dois novos arquivos dentro da nossa pasta *settings*, estes arquivos são: *local.py* e *production.py* e vamos importar tudo do nosso arquivo *base.py* com o seguinte comando: ***from .base import ****.

Se alterarmos novamente a atualização do passo anterior e mudarmos para *'ByteBuilders.settings.local'* ou para *'ByteBuilders.settings.production'* então veremos um output no terminal indicando que a nossa aplicação está utilizando as configurações definidas desses arquivos que acabamos de criar.

OBS: é importante ressaltar que essas alterações são alterações manuais e o ideal é termos isso automatizado.

Se olharmos novamente no nosso arquivo *base.py* dentro da pasta *settings*, veremos:

```
DEBUG = True
```

Isso indica que sempre que encontrarmos um erro durante a execução da nossa aplicação, o Django nos mostrará detalhadamente o que está acontecendo e porque o erro está ocorrendo. Em um ambiente de produção ter essa opção ativa é um risco enorme, e NUNCA deve ser utilizada pois devemos proteger nossas informações sensíveis e confidenciais.

Então ao automatizar a escolha entre as configurações locais ou de produção devemos primeiro ir ao nosso arquivo *manage.py* e vamos definir uma condição com uma lógica simples: se o aquela configuração for verdadeira (True) vamos executar a configuração local, se for falsa (false) vamos executar a configuração de produção.

Para implementar essa automação, no nosso arquivo *manage.py*, devemos importar o nosso arquivo base da nossa pasta settings, fazemos isso com o seguinte código: ***from ByteBuilders.settings import base***.

E então adicionamos a nossa lógica condicional:

```
    if base.DEBUG:
        os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'ByteBuilders.settings.local')
    else:
        os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'ByteBuilders.settings.production')
```

Se alterarmos agora a configuração do DEBUG para False no arquivo *base.py*, veremos um erro ao executar a nossa aplicação, pois precisamos permitir todos os hosts. Fazemos isso realizando a seguinte alteração: ***ALLOWED_HOSTS = ['*']***, se executarmos a nossa aplicação novamente, veremos que a configuração de produção será selecionada automaticamente, e se colocarmos a configuração do DEBUG para True veremos a configuração local sendo selecionada.

Secret Key

Por padrão, o Django cria uma key (chave) para a gente (obs: repare no início django-insecure).
A chave secreta deve ser única para cada aplicação do Django pois ela fornece uma assinatura criptografada.

Existem várias ferramentas para gerar uma Secret Key, mas vamos utilizar a ferramenta que o Django fornece, para gerar a nossa própria ferramenta usaremos precisamos acessar o shell do python, no nosso caso, vamos acessar o shell da seguinte forma ***./manage.py shell***, dentro deste shell teremos acesso às ferramentas do Django… com isso vamos utilizar os seguintes comandos: ***from django.core.management.utils import get_random_secret_key*** e então vamos digitar em sequência ***print(get_random_secret_key())*** e então isso irá gerar a nossa Secret Key, então podemos copiá-la e substituí-la na nossa aplicação.

Variáveis de ambiente

pip install python-dotenv
https://pypi.org/project/python-dotenv/

Para deixar a nossa aplicação mais segura, vamos utilizar dotenv, que é uma ferramenta que nos permite adicionar toda nossa informação sensível dentro de um ambiente local e então acessa-la por toda a nossa aplicação Django.

Após a instalação do dotenv, devemos ir ao root da nossa aplicação e criar um arquivo chamado ***.env*** e então vamos definir o que queremos listar.
No nosso caso, queremos armazenar a nossa Secret Key e DEBUG que estão no arquivo *base.py*, e então, vamos realizar algumas alterações no nosso arquivo de configuração

base.py para que ele possa ler o arquivo que acabamos de criar *.env*. Então no topo do nosso arquivo *base.py* vamos colocar o seguinte código: ***from dotenv import load_dotenv***. Isso irá chamar o nosso pacote e então irá carregar as nossas variáveis de ambiente e então podemos acessá-las através do nosso arquivo de configuração. É necessário também acessá-las através do nosso sistema operacional, devemos adicionar também ***import os*** e por fim, devemos adicionar o load_dotenv() para carregar as nossas variáveis de ambiente

Ficando assim:

```
from pathlib import Path
from dotenv import load_dotenv
import os

load_dotenv()
```

Já nas variáveis SECRET_KEY e DEBUG devemos colocar da seguinte forma:

```
SECRET_KEY = os.environ.get('SECRET_KEY')
DEBUG = os.environ.get('DEBUG')
```

Se nós executarmos o nosso servidor ele irá sempre executar como 'local' independente se o nosso DEBUG for True ou False no arquivo *.env*.
Isso irá ocorrer pq o nosso DEBUG é booleano e no momento que retornarmos uma string uma 'não string' ou string vazia vai retornar True, então essa nossa condição sempre irá retornar verdadeira pois a nossa variável de ambiente, retorna uma String para nós.
O que precisamos fazer para corrigir isso é adicionando um operador de igualdade para verificar se a nossa String é True, se a variável de ambiente retornar False então ela não será igual a verdadeira portanto a nossa lógica aqui retornará False e assim poderemos definir se vamos iniciar a nossa aplicação no modo local ou de produção.

O código atualizado ficará assim:

```
DEBUG = os.environ.get('DEBUG') == 'True'
```

Testes

pip install pytest
https://pypi.org/project/pytest/

Existem diferentes frameworks para realizar testes. Nós iremos utilizar o Pytest, este framework nos fornece diversos plugins e extensões.

Nós iniciamos o Pytest apenas digitando no terminal: *pytest*.
Na pasta raiz do nosso projeto onde está o arquivo *manage.py*, vamos criar um arquivo chamado *pytest.ini*, será um arquivo que será iniciado quando executarmos o pytest.

Dentro deste arquivo, vamos colocar o seguinte código:

```
[tool:pytest]
DJANGO_SETTINGS_MODULE=ByteBuilders.settings.local
python_files=test_*.py
```

Sempre quando formos realizar algum teste, devemos digitar no terminal:

```
export DJANGO_SETTINGS_MODULE=ByteBuilders.settings.local
```

Define a variável de ambiente *DJANGO_SETTINGS_MODULE* para que aponte para o arquivo de configurações *local.py* dentro do módulo *ByteBuilders.settings*.
Depois de definir a variável de ambiente dessa forma, devemos ser capaz de executar o pytest sem problemas, desde que esteja no diretório correto do projeto Django (onde o arquivo *manage.py* está localizado) e seu ambiente virtual esteja ativado.Agora digitamos pytest no terminal para verificar se deu tudo certo.

Após a realização dos testes, devemos digitar novamente no terminal o seguinte comando:

```
unset DJANGO_SETTINGS_MODULE
```

Seguindo o conceito da modularização que realizamos anteriormente, vamos criar uma pasta chamada tests.

'Observação:
Isso pode ser definido de diversas formas, diversos projetos podem definir os seus testes dessa maneira ou de outra maneira. É uma escolha definida na hora do desenvolvimento, seja pessoal ou do time de desenvolvimento em questão.
A nossa abordagem aqui será dessa maneira, sintam-se livres para explorar outras formas de testes no ambiente pessoal de vocês.'

Essa pasta estará localizada dentro da nossa aplicação principal, e então criaremos um novo arquivo seguindo o padrão que foi definido no arquivo *pytest.ini*, todo arquivo de teste criado deve começar com ***"test_*alguma_coisa*"***, vamos criar um arquivo de teste para vermos se a nossa configuração está funcionando corretamente, dentro deste arquivo vamos colocar o seguinte código que será uma função:

```
def teste_example():
    assert 1 == 1
```

O assert basicamente irá retornar True or False, nós vamos comparar duas coisas, se retornar True significa que o nosso teste funcionou, e se retornar False significa que o nosso teste falhou, em algumas situações, podemos configurar o nosso teste para falhar propositalmente para verificarmos se o que implementamos está correto, mas veremos mais disso no decorrer do curso.

Git

Primeiro vamos criar uma conta no github e um repositório.
Após a criação da conta e do repositório vamos no terminal e vamos digitar dois comandos para gerar e recuperar uma chave para conexão SSH.

Este primeiro comando irá gerar uma chave SSH: ***ssh-keygen -t rsa -b 4096 -C "your_email@example.com"***
Este segundo comando irá exibir a segunda chave SSH: ***cat ~/.ssh/id_rsa.pub***

Após termos criado e exibido a chave SSH, devemos ir ao github, clicar na foto do nosso perfil, depois em settings e depois em SSH e GPG keys, clicar em “New SSH key”, adicionar um título e adicionar a chave no campo “Key”.

E agora vamos criar um novo repositório na nossa conta do github, e em seguida vamos criar um arquivo na nossa pasta principal chamado *.gitignore* e vamos adicionar alguns itens a serem ignorados:

```
venv/
.pytest_cache/
__pycache__/
db.sqlite3
.env
```

Após fazer todas as etapas anteriores, vamos enviar o nosso código para o nosso repositório principal ou “main”.
Para isso, vamos utilizar os seguintes comandos:
Aqui iniciamos o nosso repositório local

```
git init
```

Vinculamos nosso repositório com o do github

```
git remote add origin git@github.com:FilipeFerCosta/ByteBlog.git
```

Colocamos para rastrear todos os arquivos (que não estão no arquivo *.gitignore*)

```
git add .
```

Entramos com as nossas informações

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Damos o primeiro ‘commit’ do nosso código

```
git commit -m "Initial commit"
```

Definimos o branch que queremos usar

```
git branch -M main
```

Enviamos para o github para o branch definido

```
git push -u origin main
```

Configurando algumas extensões no VS Code

Há diversas formas de configurar o VS Code, podemos ir nas configurações e personalizar da forma que quisermos, entretanto, podemos utilizar essa IDE para trabalhar com diversas linguagens, e realizar esse tipo de configuração ela é aplicada globalmente.
Portanto, vamos abordar aqui uma forma de mitigar essa questão, que será utilizar extensões para realizar nossa configuração o mais flexível possível.

- Black Coder Formatter - Integração
    - https://pypi.org/project/black/
    - pip install black

Há duas formas de instalar essa extensão, uma delas é através da loja de extensões do VSCode, a outra é através do pip e configurar manualmente. Nós iremos instalar via pip e configurar manualmente.

Após instalar o black, vamos criar uma pasta chamada *.vscode* na pasta global e um arquivo dentro desta pasta chamado *settings.json* e vamos adicionar o seguinte código:

```
{
    "editor.formatOnSave": true,
    "python.formatting.provider": "black",
    "python.formatting.blackArgs": [
        "--line-length",
        "88"
    ],
    "[python]": {
        "editor.codeActionsOnSave": {
            "source.organizeImports": true
        },
        "editor.defaultFormatter": "ms-python.black-
formatter"
    }
}
```

Posteriormente vamos criar outro arquivo na pasta global chamado *vscode.md* e adicionar este código lá também.
Vamos fazer isso pois a nossa pasta .vscode entrará no .gitignore.

- Flake8
    - https://pypi.org/project/flake8/
    - pip install flake8

Entre os dois códigos do arquivo settings.json adicione:

```
    "python.linting.enabled": true,
    "python.linting.lintOnSave": true,
    "python.linting.flake8Enabled": true,
```

```
    "python.linting.flake8Args": [
        "--max-line-length",
        "88"
    ],
```

Essa configuração irá nos mostrar erros que não são necessariamente erros, mas configurações, formatação não adequada ou pedaços de código não utilizados e que não deveriam estar ali.

Review do que foi feito até o momento

Até o presente momento, instalamos aplicações dos pacotes do python e nós criamos o nosso projeto base, o nosso framework e o nosso ambiente para testes e desenvolvimento da nossa aplicação.

Agora queremos replicar tudo isso, porque se a gente passar esse código para outra pessoa, ou para outro computador, ele não vai funcionar porque não terão todos os pacotes instalados.
Por exemplo, nós instalamos o Django, Flake 8, Black, etc.
Nós precisamos manter um registro de tudo isso, então o que vamos fazer é utilizar um arquivo de requerimentos.
Vamos utilizar o pip para criar esse arquivo para nós, faremos isso com o seguinte comando: ***pip freeze > requirements.txt***

Agora se quisermos mover o nosso código para outro ambiente ou compartilhá-lo com alguém, tudo que vamos precisar fazer é: ***pip install -r requirements.txt*** e o pip irá instalar todos os pacotes necessários no novo ambiente.
É importante mencionar que devemos utilizar esse comando no diretório que queremos salvar o arquivo *requirements.txt*.

Criando o primeiro APP

Nós queremos que a nossa aplicação esteja dentro do nosso diretório (pasta) principal, que é onde estão as nossas pastas *settings* e *tests*.
Então vamos criar uma pasta dentro desse diretório principal chamada *blog* e agora vamos executar o seguinte comando ***./manage.py startapp blog ./ByteBuilders/blog***.

Agora vamos registrar o nosso novo app, para que o Django saiba que ele existe. Para fazer isso, devemos ir para a pasta *settings*, no arquivo *base.py* devemos procurar por ***"INSTALLED_APPS"*** e adicionar ***"ByteBuilders.blog",*** e como colocamos a nova aplicação dentro da aplicação principal, precisamos editar o arquivo *apps.py* que está dentro da nova pasta *blog* e editar o código que está lá para ficar assim:

```
from django.apps import AppConfig


class BlogConfig(AppConfig):
```

```
    default_auto_field = "django.db.models.BigAutoField"
    name = "ByteBuilders.blog"
```

Versionamento de código

Versionamento de código é uma ferramenta poderosa quando estamos trabalhando em equipes ou quando vamos lançar ‘releases’ ou versões diferentes da nossa aplicação. Podemos separar o nosso desenvolvimento em ‘módulos’ sem necessariamente alterar o nosso projeto principal (main project).

Isso nos dá uma série de benefícios, como:

- Controle de versão o que nos permite rastrear e saber quem realizou mudanças;
- Colaboração Eficiente, permitindo que várias pessoas trabalhem em equipe no mesmo projeto simultaneamente sem conflitos;
- Testes e Validação, que nos permite a criação de ramos (branches) separados para desenvolver e testar novos recursos ou correções de bugs sem afetar a versão principal;
- Rollbacks Seguros, pois se algo der errado, é possível voltar facilmente para uma versão anterior do código, minimizando o impacto de erros ou problemas;
- Gestão de Releases que nos ajuda a gerenciar diferentes versões da aplicação, permitindo a criação de branches específicas para cada versão e a manutenção de correções de segurança em versões mais antigas;
- Isolamento de Módulos: Como mencionou, é possível separar o desenvolvimento em módulos ou componentes independentes, o que torna mais fácil trabalhar em partes específicas do projeto sem afetar o todo;
- Histórico de Alterações que nos ajuda a fornecer um histórico detalhado de todas as alterações feitas ao longo do tempo, ajudando a entender a evolução do projeto;
- Padrões de Desenvolvimento que pode ser usado para impor padrões de desenvolvimento e boas práticas, garantindo consistência no código.

Como estamos utilizando o GitHub, devemos ir para o repositório do nosso projeto, e vamos clicar em main, vermos ali apenas um Branch, que é o nosso Branch principal e então vamos clicar em ***main***, ***View*** all branches e então ***New branch*** e vamos chamar o nosso branch de *project-v1.0.1* e se voltarmos ao main, veremos o nosso novo branch.

Agora vamos enviar o nosso código para o novo branch seguindo os seguintes passos:
Mas antes vamos confirmar se o novo branch existe remotamente, fazemos isso com o seguinte comando no terminal: ***git ls-remote --heads origin***
Após a verificação, vamos mudar para o nosso novo branch com o seguinte comando: ***git checkout -b project-v1.0.1 origin/project-v1.0.1*** este comando irá criar localmente um novo branch e o origin irá especificar qual branch remoto vamos utilizar, então vamos usar o comando ***git add .*** para rastrear novamente todos os arquivos e o comando ***git commit -m "Sua mensagem de commit aqui"*** para comitar as nossas alterações e por fim o comando ***git push origin project-v1.0.1*** para enviar as nossas alterações para o novo branch.

Agora vamos verificar no github o envio do nosso projeto para o novo branch.

Criando modelo de dados

Os modelos (models) eles descrevem as tabelas do nosso banco de dados, e como vamos construir um blog, nós vamos precisar apenas de uma tabela que será a estrutura dos nossos posts, então vamos essa tabela e chamá-la de *post*. Para isso, devemos ir na pasta *blog* do novo app criado, e selecionar o arquivo *models.py*, e por se tratar de uma classe, devemos usar a letra inicial em maiúsculo. As classes são um modelo de objeto, e cada publicação feita será uma instância (objeto) dessa classe.
Ao declarar a nossa classe, vamos herdar (ou extender) da classe ***moldels.model*** para podermos utilizar as ferramentas disponibilizadas pelo Django e definir o nosso modelo.

```
class Post(models.Model):
```

Agora vamos por partes para o nosso modelo de dados.
Vamos começar definindo um título e subtítulo:

```
    titulo = models.CharField(max_length=250)
    subtitulo = models.CharField(max_length=100)
```

"models.CharField" é usado para campos de texto curtos.
max_length=250 é um parâmetro que define o comprimento máximo do campo. Neste caso, o título da postagem deve ter no máximo 250 caracteres e o subtítulo 100.

Agora uma Slug:

```
    slug = models.SlugField(max_length=250, unique=True)
```

Uma slug é uma versão amigável para URL de um texto, geralmente usado para representar títulos ou nomes em um formato legível, com espaços substituídos por hífens ou traços sublinhados e caracteres especiais removidos.
E como vamos utilizar o slug para navegar em duas postagens individuais, vamos ter que transformar isso em algo único, para não termos duas Slugs iguais.

Agora o autor do post:

```
    autor = models.ForeignKey(User,
on_delete=models.CASCADE, related_name="autor_post")
```

Quanto ao autor, ou usuário, também precisa ser único, mas como ainda não vamos implementar um modelo de usuário, vamos utilizar o básico do Django.
Vamos importar o modelo de usuário padrão fornecido pelo Django, o Django tem um sistema de autenticação embutido e o modelo de usuário é uma parte fundamental desse sistema, fazemos isso com o seguinte código ***from django.contrib.auth.models import User***.
E se um usuário for deletado, qualquer publicação que este usuário tenha feito, deverá ser apagada em junto com ele fazemos isso com o seguinte código

on_delete=models.CASCADE, nós podemos mudar isso posteriormente, mas vamos deixar dessa forma, vamos adicionar também ***related_name="autor_post"*** para quando quisermos executar alguma query ou uma query reversa nós vamos utilizar o related name para identificar este dado, nós chamamos de *autor_post*.

Temos também o conteúdo do nosso post, que será uma área de texto.

```
    conteudo = models.TextField()
```

Vamos ter a hora e o dia em que a publicação foi criada:

```
    criado_em = models.DateTimeField(auto_now_add=True, editable=False)
```

Mas não vamos inserir isso manualmente, vamos deixar o django realizar essa tarefa para nós, definimos isso com o seguinte código: ***auto_now_add=True***. Nós também não queremos essa sessão editável, então adicionamos o seguinte código: ***editable=False***.

Teremos também um campo para mostrar quando a publicação foi editada (atualizada):

```
    atualizado_em = models.DateTimeField(auto_now=True)
```

Quando editarmos (ou atualizarmos) as nossas publicações, devemos ter também atualizado sempre a data e a hora da edição, podemos deixar ou não editável.

Vamos utilizar o método "Dunder string" que é um método mágico (especial), alguns exemplos de métodos mágicos são ***__init__***, ***__str__***, ***__repr__***...
O método ***__str__*** é usado para definir a representação de string legível para humanos de um objeto em Python. Quando você chama a função ***str()*** em um objeto ou quando utiliza o operador de concatenação ***'+'*** com uma string, o Python procura e executa o método ***__str__*** desse objeto, retornando a representação de string resultante.

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

Vamos adicionar também uma classe de metadados para ordenar os nossos posts de acordo com a data de criação

```
    class Meta:
        ordering = ("-criado_em",)
```

E por fim vamos adicionar mais um campo para determinar se a nossa publicação está publicada ou não (visível ou não no site).

```
    status = models.CharField(max_length=10,
choices=options, default="draft")
```

Então devemos especificar também as opções da nossa variável options.

```
    options = (
        ("draft", "Rascunho"),
        ("published", "Publicado"),
    )
```

Nosso modelo no final então, ficará da seguinte forma:

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


class Post(models.Model):
    options = (
        ("draft", "Rascunho"),
        ("published", "Publicado"),
    )

    titulo = models.CharField(max_length=250)
    subtitulo = models.CharField(max_length=100)
    slug = models.SlugField(max_length=250, unique=True)
    autor = models.ForeignKey(User,
on_delete=models.CASCADE, related_name="autor_post")
    conteudo = models.TextField()
    criado_em = models.DateTimeField(auto_now_add=True,
editable=False)
    atualizado_em = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
choices=options, default="draft")

    class Meta:
        ordering = ("-criado_em",)

    def __str__(self):
```

```
        return self.titulo
```

Migração do modelo para o Banco de Dados

Como mencionado anteriormente, a nossa model(modelo) é uma descrição ou modelo de um banco de dados.
Sempre que realizarmos mudanças ou criar algum novo modelo de dados, nós devemos realizar essa migração para atualizar ou até mesmo criar o nosso banco de dados de acordo com o nosso modelo.

Para realizar a migração nós devemos digitar os seguintes códigos no terminal:

- ***./manage.py makemigrations*** este comando fará o Django inspecionar todas as mudanças feitas e então criará uma pasta chamada 'migrations' e terá um arquivo chamado *0001_initial.py*, o Django criou um script do nosso modelo de dados e armazena esse script neste arquivo que será executado no banco de dados. Então basicamente o Django está traduzindo o nosso código python para que o nosso banco de dados possa utilizar, ou seja linguagem SQL.

Antes de aplicar o último comando no terminal, se nós executarmos o ***./manage.py runserver***, veremos essa mensagem que está nos dizendo que temos migrações não aplicadas no nosso projeto.

- Agora vamos fazer a nossa migração com o seguinte código: ***./manage.py migrate***. Em seguida, para visualizarmos o nosso banco de dados criado por padrão pelo Django, precisamos instalar o sqlite3 no nosso sistema, e posteriormente, procurar pela extensão SQLite e instalá-la.

Podemos notar que no nosso modelo, nós não especificamos uma chave primária (primary key), como não mencionamos, o Django por padrão cria uma para nós.

E na nossa pasta migrations que está dentro da pasta blog, ela funciona um pouco similar com o versionamento de código, nós temos um histórico das alterações feitas e isso nos permite retornar a estados anteriores da nossa tabela e também nos permite acompanhar as mudanças que fizemos.

Se quisermos 'resetar' tudo, basta deletar o arquivo *db.sqlite3* e o arquivo *0001_initial.py* e então executar o processo novamente da migração para recriá-los.

Painel administrativo

Vamos acessar o admin painel, para isso precisamos registrar um super usuário no nosso projeto, fazemos isso com o seguinte código: ***./manage.py createsuperuser***. Este comando criará um usuário administrador.
Agora vamos acessar o painel administrativo (admin panel), para isso devemos iniciar o nosso servidor e na url de acesso à aplicação, devemos adicionar /admin no final, ficando assim: ***http://127.0.0.1:8000/admin/***.

Vale notar que não temos a nossa área de publicações (Post), nós criamos o nosso modelo de dados, mas não adicionamos ele no painel administrativo do Django.
Para isso, devemos voltar ao nosso código na pasta *blog* e abrir o arquivo chamado *admin.py* e adicionar o nosso modelo (model), isso é feito da seguinte forma:

Primeiro devemos importar o nosso modelo com o seguinte código:

```
from .models import Post
```

Agora precisamos registrá-lo, e isso é feito com o seguinte código:

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

O que deve deixar o código no arquivo da seguinte forma:

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

admin.site.register(Post)
```

Se voltarmos agora à área administrativa do Django, veremos a nossa área de publicações (Post).

Vamos criar uma nova publicação e vamos ver se tudo ocorreu corretamente.
Se não houver nenhum erro, a nova publicação deve aparecer em lista com o título que escolhemos.

Teste do modelo de dados

Agora que construímos nosso modelo de dados, vamos implementar um teste para ver se ele está funcionando perfeitamente.

Podemos fazer testes para verificar todo o modelo ou partes dele.

Lembra daquela pasta *tests* que criamos? Pois bem, vamos utilizá-la novamente.
Dentro desta pasta, vamos criar uma nova pasta chamada *blog*, queremos que essa pasta seja tratada como um módulo, portanto vamos ter um arquivo dentro dela chamado *__init__.py* igual mencionado anteriormente quando criamos o nosso projeto e realizamos a nossa modularização.

Lembra quando definimos o escopo do módulo de testes e a configuração que fizemos? Aquelas regras também valem para a nova pasta *blog* que criamos dentro da pasta *tests*, portanto vamos criar um novo arquivo chamado *test_models.py* dentro da pasta *blog* e instalar o *pytest-django* com o seguinte comando: ***pip install pytest-django*** (https://pypi.org/project/pytest-django/).

Após isso, vamos utilizar o *pytest* e o *pytest-cov* para saber quais testes são necessários, nós já temos o *pytest* instalado, então falta instalar o *pytest-cov*, fazemos isso com o seguinte comando no terminal: ***pip install pytest-cov*** (https://pypi.org/project/pytest-cov/). E para utilizar o pytest-cov devemos digitar no terminal: ***pytest --cov***, com isso veremos uma indicação de onde é necessário realizar testes.

Nós podemos gerar um relatório também, fazemos isso com o seguinte código no terminal: ***pytest --cov-report html --cov=./*** com este código, especificamos para criar um relatório no diretório atual do nosso terminal, que no nosso caso é a nossa pasta raiz.
Vamos adicionar essa nova pasta *htmlcov* para o *.gitignore* pois não queremos ela no nosso repositório por conter informações sensíveis.
Agora para inspecionar o nosso relatório, vamos abrir o arquivo *index.html* que está dentro da pasta gerada com o nosso navegador, e poderemos inspecionar com mais detalhes onde os testes são necessários.

É importante mencionar que quando rodamos geramos o nosso relatório completo na pasta *htmlcov*, há bem mais detalhes do que o comando simples que digamos no terminal com ***pytest --cov***.

Vamos imaginar os testes em uma aplicação enorme, com variadas funções, então poderíamos ter centenas ou talvez até milhares de testes e estes testes usariam a nossa base de dados (banco de dados) e isso poderia deixar a nossa aplicação ou teste extremamente pesados ou até mesmo comprometer os dados da nossa base de dados, para evitar isso vamos criar algumas factories que são basicamente pedaços de código que podemos reutilizar sempre quando vamos realizar esses testes que requerem dados da nossa base de dados.

Então vamos na nossa pasta *tests* e vamos criar um arquivo chamado *factories.py* e dentro desse arquivo, nós vamos construir os nossos testes.

Nós vamos precisar utilizar algumas ferramentas aqui:

A primeira delas será factory boy, instalamos essa ferramenta com o seguinte código: ***pip install factory-boy*** (https://pypi.org/project/factory-boy/). Essa ferramenta nos ajudará a gerar dados e irá criar os 'factories' para nós.

Agora, dentro do nosso arquivo factories.py vamos começar a construir o nosso teste.

Vamos destrinchar os seguintes códigos que irão compor o nosso arquivo:

Vamos começar importando o factory:

```
import factory
```

E o que precisamos fazer aqui é preparar o código para inserir dados na nossa base de dados e sempre quando rodamos o nosso teste. E para isso vamos precisar das nossas tables, fazemos isso com o seguinte importe:

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

Estamos chamando o nosso modelo de usuários que vem por default no Django.

E vamos precisar também do nosso modelo de dados da classe *Post* que criamos dentro do nosso arquivo *models.py* que está dentro da nossa pasta *blog*.
Fazemos isso com o seguinte código:

```
from ByteBuilders.blog.models import Post
```

Então agora podemos criar a nossa primeira factory, vamos chamar essa factorie de *Post*, nós vamos utilizar algumas ferramentas do django aqui tbm, então vamos criar a nossa classe chamada *PostFactory* e vamos utilizar o *DjangoModelFactory*, isso significa que podemos utilizar que já está escrita na nossa model como mencionamos anteriormente no import para determinar quais dados criar.

```
class PostFactory(factory.django.DjangoModelFactory):
```

Depois vamos criar uma nova classe dentro da classe *PostFactory* chamada *Meta*, e dentro dela vamos especificar o modelo que vamos utilizar que no nosso caso é o *Post*, ficando assim:

```
class PostFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Post
```

E agora vamos começar a construir alguns dados do nosso modelo

```
    titulo = "x"
    subtitulo = "x"
    slug = "x"
    autor =
```

Mas aqui encontramos um problema, nós precisamos construir antes uma factory pois não conseguimos conectar o nosso post a um autor sem uma foreign key (chave estrangeira) de um autor.
Então, sempre que criarmos um novo post, vamos ter que criar um novo autor, a classe do autor ficará assim:

```
class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = User

    password = "test"
    username = "test"
    is_superuser = True
```

```
    is_staff = True
```

Se olharmos no nosso banco de dados, na tabela *auth_user* poderemos ver os campos que preenchemos ali como os dados do nosso modelo de usuário, alguns dados são requisitos básicos outros não.

Então continuando com o nosso modelo *Post*, a parte do autor ficará assim:

```
    autor = factory.SubFactory(UserFactory)
```

Especificamos aqui para usar a classe que acabamos de usar a *UserFactory*, então sempre que rodarmos esse teste, ele vai criar um novo usuário, e vai 'linkar' este novo usuário à essa publicação com o uso da chave estrangeira.

O nosso código precisa também da parte de *conteudo* e *status*, vale ressaltar que *status* tem dois estados, publicado e rascunho, vamos especificar como publicado. Não precisamos mencionar o *criado_em* nem o *atualizado_em* pois isso será gerado para nós automaticamente.
O código da classe *PostFactory* ficará assim no final:

```
class PostFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Post

    titulo = "x"
    subtitulo = "x"
    slug = "x"
    autor = factory.SubFactory(UserFactory)
    conteudo = "x"
    status = "published"
```

E o código completo, ficará assim:

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

from ByteBuilders.blog.models import Post


class UserFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = User
```

```
    password = "test"
    username = "test"
    is_superuser = True
    is_staff = True


class PostFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Post

    titulo = "x"
    subtitulo = "x"
    slug = "x"
    autor = factory.SubFactory(UserFactory)
    conteudo = "x"
    status = "published"
```

Agora que temos a nossa Factory, devemos chamá-la no nosso teste, queremos criar os dados e então rodar o nosso teste.

Agora vamos instalar a segunda ferramenta que vamos precisar utilizar:
pip install pytest-factoryboy
https://pypi.org/project/pytest-factoryboy/

E vamos para dentro da nossa pasta *test* e vamos criar um novo arquivo chamado *conftest.py*, este arquivo será visto pelo pytest e dentro deste arquivo, vamos criar uma configuração. Nós precisamos é importar o *register* do *pytest_factoryboy* precisamos fazer isso para utilizá-la no nosso teste e também importar o nosso *PostFactory* e registrá-lo, ficando assim:

```
from pytest_factoryboy import register
from .factories import PostFactory

register(PostFactory)
```

Agora ficará disponível e podemos utilizar a nossa factory no nosso teste.

Agora vamos voltar para a nossa pasta *blog* que está dentro da pasta *test* e vamos ao arquivo *test_models.py*, então o que faremos aqui é criar um novo objeto na base de dados e retornar esse objeto para ver se está tudo funcionando corretamente.
Vamos começar a construir nosso código de teste.

Vamos criar uma nova classe chamada TestPostModel:

```
class TestPostModel:
```

Depois vamos criar uma função dentro dessa classe para testar o retorno da string, ficando assim:

```
class TestPostModel:
    def test_str_return(self, post_factory):
        post = post_factory(titulo="test-post")
        assert post.__str__() == "test-post"
```

Dentro da nossa função, nós passamos como parâmetro "*post_factory*", fizemos dessa forma porque estamos dizendo ao *pytest_factoryboy* para criar uma instância da fábrica *PostFactory* sempre que usarmos o nome *post_factory* em nossos testes.

Estamos chamando a fábrica *PostFactory* usando o nome registrado *post_factory* e passando o argumento *titulo="test-post"*. Isso cria uma instância de *PostFactory* com o título especificado e a associa à variável *post*.
Dessa forma, podemos usar essa instância para realizar as asserções em nosso teste. Essa abordagem torna o código de teste mais legível e fácil de entender, pois estamos utilizando um nome mais descritivo *(post_factory)* em vez de ter que usar o nome completo da fábrica *(PostFactory)* toda vez que quisermos criar uma instância de um objeto usando essa fábrica.

Vamos fazer uma configuração para que todas as classes definidas no nosso arquivo *test_models.py* funcionem normalmente nos nossos testes.

```
import pytest

pytestmark = pytest.mark.django_db
```

Este código define um marcador (ou decorator) *pytest.mark.django_db* para marcar os testes como aqueles que requerem acesso ao banco de dados em um ambiente Django. Esse marcador é usado em testes de unidade ou testes de integração que precisam interagir com o banco de dados do Django, seja para criar registros, consultar dados ou realizar outras operações relacionadas ao banco de dados.

Quando marcamos um teste com *pytest.mark.django_db*, o Pytest garante que um banco de dados de teste seja configurado antes da execução do teste. Isso permite que você escreva testes que dependem de um banco de dados real ou simulam interações com o banco de dados sem afetar o banco de dados de produção.

Aqui estão alguns pontos-chave a serem observados sobre o uso de *pytest.mark.django_db*:

1. **Isolamento do banco de dados:** O Pytest cria um banco de dados de teste isolado para cada teste marcado com *pytest.mark.django_db*. Isso garante que os testes

não afetem o banco de dados de produção e que cada teste comece com um estado de banco de dados limpo.

2. **Rollback automático:** Após a execução de um teste marcado com *pytest.mark.django_db*, o Pytest geralmente reverte quaisquer alterações no banco de dados, garantindo que o banco de dados volte ao seu estado original após o teste.
3. **Facilita a criação de dados de teste:** Você pode usar o banco de dados de teste para criar registros de teste e realizar consultas para verificar se suas funções ou modelos estão funcionando conforme o esperado.

O código completo no final ficará assim:

```
import pytest

pytestmark = pytest.mark.django_db



class TestPostModel:
    def test_str_return(self, post_factory):
        post = post_factory(titulo="test-post")
        assert post.__str__() == "test-post"
```

Agora vamos no terminal e digitar o seguinte comando ***pytest***, nós veremos um erro de falando que não foi possível realizar o import porque não há pacotes pais conhecidos.

Isso vai acontecer porque nós não temos um arquivo *__init__.py* na nossa pasta tests, então basta criá-lo e rodar o comando ***pytest*** novamente que tudo ocorrerá bem.

Vamos executar o comando ***pytest --cov-report html --cov=./*** para ver se algum dos testes solicitados sumiu.

Branch Merge

Antes de aprendermos a realizar o merge, vamos criar um novo commit e dar um update no nosso arquivo *.gitignore*, além de atualizar o nosso *requirements.txt*.

Até o momento o nosso arquivo gitignore está assim:

```
venv/
.pytest_cache/
__pycache__/
db.sqlite3
.env
.vscode/
migrations/
```

Vamos adicionar:

```
htmlcov/
.coverage
```

Agora vamos enviar o nosso código para o branch *project-v1.0.1*, para isso vamos com os seguintes códigos:
git add .
git commit -m "Conftest configured, added a test_models.py and created a factory"
git push origin project-v1.0.1

Com tudo isso feito, podemos agora ir ao github e então acessar o repositório do nosso projeto.
Podemos ver os dos branches que criamos. Vamos simular aqui um ambiente corporativo, de produção. Imagine que finalizamos este ciclo do desenvolvimento e queremos que essa versão se torne a nossa versão principal, isso possibilitaria que outra pessoa possa pegar este novo código base e adicionar novos recursos. Então para concretizar isso, vamos realizar um 'merge' para que o nosso branch principal (main) esteja atualizado.

Vamos clicar em *Compare & pull request*, podemos ver as modificações feitas nos arquivos e então em *Create pull request*. Em um ambiente de colaboração, o desenvolvedor sênior por exemplo, receberia esse pedido de merge e poderia fazer as revisões necessárias no código e aprová-lo ou não, nessa parte.

Nós podemos 'apagar' o branch que estávamos utilizando, mas não vamos apagá-lo para manter um histórico do nosso progresso.

Vamos criar um novo branch para continuar o nosso desenvolvimento, vamos chamá-lo de ***project-v1.0.2***, o processo é o mesmo de quando criamos o ***project-v1.0.1***.

Django View

Se olharmos dentro da pasta *blog* da nossa aplicação, podemos notar a existência de um arquivo chamado *views.py*, o Django trabalha com o uso de templates, mais especificamente templates em html. Neste arquivo *views.py*, nós vamos especificar toda a lógica que irá gerenciar as requisições dos usuários, carregando os templates e os dados e retornando-os para o usuário.

Há duas abordagens para manipular as views, uma delas é views baseadas em classes e a outra é funções tradicionais. Nós vamos utilizar as views baseadas em classes.
Usando as views baseadas em classes nos permitirá utilizar os templates do django e modificá-los para os nossos propósitos e como vantagem, gastaremos menos tempo construindo a nossa view, a desvantagem é porque como o django já praticamente todo código escrito a maior parte das funções estão 'abstraídas' e num primeiro momento podemos acabar não percebendo o que está acontecendo.

A documentação do Django (https://docs.djangoproject.com/en/4.2/ref/class-based-views/generic-display/) descreve duas visualizações genéricas baseadas em classe que são projetadas para mostrar dados.

No nosso caso, vamos pegar a nossa home page como exemplo, ela deverá mostrar todos os posts do nosso blog que estão armazenados na nossa base de dados e vamos paginar eles, e a medida que o usuário navegue rolando para baixo, novos dados devem ser carregados e visualizações genéricas baseadas em classe vão nos ajudar a realizar essa tarefa.

Se quisermos mostrar apenas um item, *DetailView* é a nossa opção, agora se quisermos listar vários dados, vamos usar *ListView*.

Vamos por a mão na massa, como vamos mostrar todo o conteúdo do blog, nós vamos usar o *ListView*.
Então vamos começar construindo a nossa classe *HomeView*.

Primeiro devemos importar a *ListView* das classes genéricas do Django

```
from django.views.generic import ListView
```

E agora vamos criar a nossa classe HomeView com o seguinte código e invocando a classe genérica *ListView* do Django:

```
class HomeView(ListView):
```

Posteriormente, devemos especificar o modelo que queremos utilizar, no nosso caso é o modelo que criamos no arquivo *models.py* chamado *Post* mas antes devemos importar o nosso modelo, fazemos isso com o seguinte código:

```
from .models import Post

class HomeView(ListView):
    model = Post
```

O ponto antes de models, específica que o arquivo que contém a nossa tabela (*models.py*) está no mesmo diretório que o arquivo *views.py*.

Depois devemos especificar o nosso template, fazemos isso com o seguinte código logo após o model:

```
    template_name = "blog/index.html"
```

O conteúdo entre aspas é indicando o diretório e o arquivo que o Django irá utilizar.
E no nosso diretório raiz (pasta *ByteBuilders*) vamos criar uma pasta chamada *templates* e dentro desta pasta vamos criar uma outra pasta chamada *blog*, e dentro dela um arquivo chamado *index.html*. Não vamos colocar nenhum markup de html ou coisa do tipo, vamos simplesmente escrever “Olá Mundo!”.

Vale a pena mencionar que especificando esse ‘path’ quando definimos na nossa view o template, mas mesmo assim, precisamos especificar para o Django onde encontrar essa pasta *templates* recém criada. Para isso vamos em vamos na nossa pasta *settings* e *base.py*, em produção nos colocaríamos os nossos templates em outra pasta e definimos essa configuração no *local.py*, mas por agora vamos deixar no arquivo *base.py*, dentro do nosso arquivo *base.py* devemos então vamos procurar pela sessão de templates e vamos preencher *'DIRS': [ ],* com *BASE_DIR / "templates"*, ficando assim:

```
TEMPLATES = [
    {
        …
        'DIRS': [BASE_DIR / "templates"],
        …
    },
]
```

Se executarmos a aplicação agora, veremos que nada mudou. Isso é porque precisamos definir os urls.

URL Path

Vamos começar modularizando as URLs, nós iremos colocar as urls específicas da nossa aplicação *blog* dentro dessa mesma aplicação e então vamos estender o URL raiz (root) para incluir essas novas URLs que estamos criando.

Dentro do nosso projeto principal (pasta *ByteBuilders*) nós temos o arquivo *urls.py* que representa o URL raiz. Nós vamos abrir esse arquivo e fazer as seguintes modificações nessas partes:

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

vamos adicionar um include após o path, deixando assim:

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

E então vamos criar um novo ‘path’:

```
    path("", include("ByteBuilders.blog.urls")),
```

Deixamos as aspas vazias para definir como o primeiro argumento do caminho no módulo, isso significa que o URL padrão para a nossa aplicação será o URL raiz, ou seja, a raiz do nosso site. Qualquer URL que corresponda à raiz do nosso site será tratado pelo aplicativo especificado em "ByteBuilders.blog.urls". Essencialmente, isso faz com que todas as URLs do aplicativo estejam acessíveis a partir do URL raiz do seu site, sem a necessidade de um prefixo adicional.

No final, ficará assim:

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

```
from django.urls import path, include

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

Agora vamos dentro da pasta *blog*, e vamos criar um arquivo chamado *urls.py*, e então vamos preencher esse arquivo com parte do conteúdo do *urls.py* raiz.

O código ficará assim:

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

urlpatterns = [
    path("", views.HomeView.as_view(), name="homepage"),
]
```

O que fizemos aqui foi o seguinte, estamos importando todas as views, e especificando o nome da classe view que queremos utilizar, definimos o nome dessa classe no arquivo *views.py* da pasta *blog*, posteriormente informamos o nome que o url terá quando iniciarmos a aplicação e fomos para a 'página' específica deste URL.

Se executarmos agora a aplicação, veremos o conteúdo que colocamos no arquivo index.html.

Base Templates

Com a quebra do paradigma procedural para o paradigma orientado a objetos, houve a ascensão do princípio "DRY" (Don't Repeat Yourself), que enfatiza a importância de não repetir código. Com o Django não é diferente, nós utilizamos o princípio DRY. E com isso nós vamos criar um template base, que estará incluído em todas as páginas que a gente criar.

Dito isso, vamos ir na pasta templates que está dentro da pasta ByteBuilders, e vamos criar uma nova pasta chamada base, e dentro desta pasta vamos criar um novo arquivo chamado base.html, este arquivo terá o html markup básico que será necessário para cada template que vamos construir para cada página web que criarmos.

Um básico sobre html, cada página html começa com uma tag <html> e termina com </html> e cada página que construímos, no lado do usuário ela terá uma seção chamada head, ela se inicia com a tag <head> e finaliza com a tag </head>, o head é basicamente metadados *(Metadados desempenham um papel fundamental no contexto de páginas da web. Eles fornecem informações que descrevem e contextualizam uma página web, ajudando os mecanismos de busca, navegadores e outros sistemas a entenderem o conteúdo da página.)*, e dentro dessa tag podemos ter métodos como títulos e

outras informações sobre a página. E tudo que for ser exibido e renderizado na página para os usuários verem, deve estar contido dentro da tag body, que também deve ser iniciada e fechada igual as outras tags mencionadas. O ponto aqui é quando precisamos que esses dados sejam renderizados pelo navegador, esses metadados precisam estar lá. Isso nos permitirá não ter que reescrever o nosso código toda vez apenas reutilizando esse template toda vez que construirmos novos templates.

Nós iremos utilizar o framework bootstrap para acelerar o nosso desenvolvimento. Seguindo para o site do bootstrap (https://getbootstrap.com/), se acessarmos a parte da documentação (seção docs), vamos copiar o segundo exemplo da página quick start e vamos colocá-lo no nosso arquivo *base.html*.

Agora vamos 'herdar' esse código do arquivo *base.html* para o nosso arquivo *index.html*, fazemos isso da seguinte forma, vamos no nosso arquivo *index.html* e vamos escrever o seguinte código lá:

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

O que estamos fazendo aqui é especificando ao Django onde o arquivo *base.html* está localizado. O *"../"* específica que estamos voltando uma pasta no diretório para posteriormente entrar na pasta *base* e então acessar o arquivo *base.html*.

Mas e se quisermos adicionar mais conteúdo na nossa página? Como fazemos utilizando o template *base.html*? Simples, vamos criar um bloco de conteúdo dentro da tag *body* assim:

```
    {% block template %}
    {% endblock %}
```

Detalhe que "template" é o nome do bloco, podemos nomear o bloco da da maneira que quisermos.

E no nosso arquivo *index.html* vamos fazer uma referência a esse bloco, da seguinte forma:

```
{% block template %}
Código aqui
{% endblock %}
```

Vamos criar um bloco também para o título, se vocês repararem, toda página da web que acessamos, fica um título na aba do navegador, isso é definido pelo título, e como o nosso template base também possui esse título, vamos definir um bloco para ele, dentro da tag head, vamos deixar essa parte assim:

```
    <title>{% block title %} Home Page {% endblock %}
</title>
```

Se atualizarmos a página agora, veremos o título da aba como "Home Page".

Mas queremos personalizar isso para cada página que acessarmos, então no nosso arquivo *index.html* vamos colocar o seguinte bloco para ser sobrescrito:

```
{% block title %}Index Page{% endblock %}
```

Se atualizarmos novamente, veremos o bloco do arquivo *index.html* sobrescrevendo o bloco do arquivo *base.html* e o título da aba será Index Page.

Podemos replicar isso para cada página/template que quisermos criar.

Static Files

Static files são arquivos que não irão mudar na nossa aplicação, eles são estáticos. Os arquivos estáticos podem incluir CSS Stylesheets, arquivos JavaScript, imagens, arquivos de fontes, arquivos de mídia, arquivos de dados, entre outros.

Nós precisamos construir na nossa aplicação algum lugar onde podemos armazenar os nossos arquivos estáticos que vão poder ser servidos da nossa página web para o usuário, seja para estilizar a página ou adicionar código que será necessário pelo nosso site para funcionar corretamente.

Vamos criar uma pasta chamada *static* dentro da pasta *ByteBuilders*, e dentro desta pasta vamos criar uma outra pasta chamada *css* e então dentro desta pasta vamos criar um arquivo CSS chamado *main.css*. Nós vamos precisar dizer ao Django onde encontrar a pasta *static*, fazemos isso indo ao arquivo *base.html* que está na pasta *base* dentro da pasta *templates* e então a vamos adicionar dentro da tag head o seguinte código:

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

<link> é uma tag HTML usada para definir relações entre o documento HTML atual e um recurso externo, no nosso caso o path até o documento main.css.

rel="stylesheet" é atributo que especifica a relação entre o documento HTML atual e o recurso vinculado. Neste caso, ele indica que o recurso vinculado é uma folha de estilos (stylesheet).

ref="{% static 'css/main.css' %}" é a parte que usa sintaxe de modelo do Django. Não é HTML padrão, e está gerando dinamicamente a URL da folha de estilos.
E ***{% static 'css/main.css' %}*** é uma tag de modelo que, quando processada pelo mecanismo de modelos do Django, gerará o path para o arquivo CSS estático.

Vamos colocar o seguinte código dentro do nosso arquivo *main.css*:

```
body {
    background-color: blueviolet;
}
```

E no topo do nosso arquivo *base.html*, precisamos colocar o seguinte código:

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

Precisamos fazer isso para carregar os arquivos estáticos no nosso código.

Abrir o modo de inspeção para mostrar como acessar o css do projeto (os arquivos estáticos) através do stylesheet como foi definido no arquivo base.html e fazer o paralelo com o css do bootstrap, MOVER A PASTA STATIC PARA OUTRO DIRETÓRIO PARA NÃO ENCONTRA-LÁ.

Não podemos acessar no momento o nosso arquivo estático, mas antes de explicar o porquê, é preciso entender uma coisa antes…

Agora quanto a configuração vamos verificar algumas questões e realizar alguns passos, na nossa pasta *settings* e no arquivo *base.py* vamos verificar se o ***staticfiles*** está na seção "INSTALLED_APPS", posteriormente

E então vamos rolar para o fim da página, veremos:

```
STATIC_URL = 'static/'
```

Essa parte do código está dizendo para o Django, para procurar a pasta *static* na pasta principal do nosso projeto (*ByteBuilders*), posteriormente quando formos colocar o nosso projeto em produção, este não será um local ideal para armazenar a pasta *static*, faremos isso por hora porque se olharmos aqui no nosso arquivo *base.py*, nessa parte em específico:

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

Basicamente diz que a nossa pasta raiz (principal) é a pasta *ByteBuilders*, se quisermos uma pasta anterior a essa, devemos adicionar mais um **.parent** ao código.
O que podemos fazer é especificar ao Django onde encontrar os arquivos estáticos que vamos criar/definir, fazemos isso da seguinte forma ao final do arquivo *base.py* com o seguinte código:

```
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Agora se executarmos o código novamente, veremos que o Django está usando o css que definimos no arquivo *main.css* que está na pasta *static*.

Bootstrap Navbar

Vamos criar um novo arquivo dentro da pasta *base* que está na pasta *templates* chamado de *navbar.html*. Então vamos ao site do bootstrap, em seguida acessar a área docs e então navbar, vamos copiar o primeiro exemplo e colocar o código no arquivo que acabamos de criar e vamos modificá-lo às nossas necessidade.

Mas antes de modificá-lo vamos ao nosso arquivo *index.html* dentro da nossa pasta *blog*, e vamos fazer o mesmo procedimento que fizemos com o *base.html*, mas dentro do bloco template, ficando assim:

```
{% block template %}
{% include "../base/navbar.html" %}

Código aqui
{% endblock %}
```

Se executarmos a aplicação, veremos a navbar que copiamos funcionando aparecendo. Como sabemos que o nosso arquivo *main.css* da pasta *static* está funcionando corretamente, vamos apagar o conteúdo que está presente ali (posteriormente voltaremos nessa pasta), talvez seja necessário realizar um hard refresh para a página atualizar sem o css.

Vamos editar o nosso arquivo *base.html* e posteriormente o *index.html*, precisamos retirar o conteúdo que não vamos utilizar, então dentro do bloco body no *base.html* vamos apagar o ***'Hello, world'***, e no *index.html* vamos apagar o ***'Código aqui'***.

Se voltarmos agora ao arquivo navbar.html, vamos ver que há várias tags representando a estrutura da nossa página.
Dentro dessas tags temos algumas classes e dentro dessas classes temos nomes de coisas separados por espaços, estes nomes correspondem a estilos que já foram criados pelo bootstrap. E se observamos bem o navbar, ele se conecta ao css, se voltarmos ao navegador e entrarmos no modo de inspeção, na seção head, podemos dar uma olhada ao css do bootstrap através do link que está contido ali, se entrarmos no modo pesquisa e pesquisar por navbar, vamos poder ver o css que corresponde ao que está acontecendo e estilizando a nossa página. Nós não estamos usando isso tudo aqui, mas alguma dessas classes corresponde ao que estamos utilizando, mas cada um desses nomes corresponde a uma classe e cada uma dessas classes está relacionada a um estilo já implementado a algum elemento específico na qual a navbar está aplicada.

Podemos ver neste exemplo:

```
<nav class="navbar navbar-expand-lg bg-body-tertiary">
```

A navbar que estamos utilizando está usando o navbar-expand-lg e bg-body-tertiary. Portanto a importância de aprender o bootstrap é para aprender cada uma dessas classes diferentes e o que elas podem nos proporcionar para estilizar a nossa página.

Vamos começar a fazer algumas alterações na navbar, se observarmos aqui nessa parte do código:

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

O que esse ***'container-fluid'*** faz é basicamente manter a navbar bem coladinha nas bordas da página, se removermos o ***-fluid***, veremos que ela ficará mais centralizada, com um espaço entre a navbar e a borda. Nós vamos deixar sem o ***-fluid***.

Esta parte do código:

```
      <button class="navbar-toggler" type="button" data-
bs-toggle="collapse" data-bs-
target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false"
aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
```

Irá mostrar um botão para as demais opções da navbar quando entrarmos na navegação mobile. (se dermos um zoom profundo, poderemos ver esse botão.

Já nessa parte do código aqui, vamos apagar todos os ‘li’:

```
      <div class="collapse navbar-collapse"
id="navbarSupportedContent">
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
          <li class="nav-item">
            <a class="nav-link active" aria-
current="page" href="#">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="#">Link</a>
          </li>
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#"
role="button" data-bs-toggle="dropdown" aria-
expanded="false">
              Dropdown
            </a>
            <ul class="dropdown-menu">
              <li><a class="dropdown-item"
href="#">Action</a></li>
              <li><a class="dropdown-item"
href="#">Another action</a></li>
              <li><hr class="dropdown-divider"></li>
              <li><a class="dropdown-item"
href="#">Something else here</a></li>
            </ul>
          </li>
          <li class="nav-item">
            <a class="nav-link disabled" aria-
disabled="true">Disabled</a>
          </li>
        </ul>
        <form class="d-flex" role="search">
```

```
          <input class="form-control me-2" type="search"
placeholder="Search" aria-label="Search">
          <button class="btn btn-outline-success"
type="submit">Search</button>
        </form>
      </div>
```

Deixando o código assim:

```
      <div class="collapse navbar-collapse"
id="navbarSupportedContent">
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">

        </ul>
        <form class="d-flex" role="search">
          <input class="form-control me-2" type="search"
placeholder="Search" aria-label="Search">
          <button class="btn btn-outline-success"
type="submit">Search</button>
        </form>
      </div>
```

Vamos manter apenas o espaço ul, pois podemos adicionar algo aqui posteriormente.

Adicionando fontes do google no website

Primeiro passo é ir ao site google fonts (https://fonts.google.com), vamos pesquisar pela fonte archivo e selecionar a opção black 900, vamos pegar a versão stylesheet e o CSS rules.
Depois vamos colocar no nosso arquivo *main.css* que está no seguinte diretório *static* > *css*.
Ficando assim por enquanto:

```
@import url('https://fonts.googleapis.com/css2?
family=Archivo:wght@900&display=swap');
```

Vamos construir também uma ‘entrada’ para a nossa logo e adicionar o CSS rules dentro dessa ‘entrada, ficando assim:

```
.logo{
```

```
    font-family: 'Archivo', sans-serif;
}
```

Dentro desse espaço vamos adicionar palavras chave na nossa classe e elas corresponderão a algo que representa um estilo.

Vamos voltar à nossa navbar, que está dentro da pasta *templates* > *base* e então *navbar.html*, dentro deste arquivo, vamos nessa parte do código:

```
<a class="navbar-brand" href="#">Navbar</a>
```

E vamos adicionar a palavra logo logo após brand, ficando assim:

```
<a class="navbar-brand logo" href="#">Navbar</a>
```

Essa parte do código em específico está conectada com a área onde vamos mostrar a nossa logo.

Agora vamos voltar à nossa aplicação e vamos executá-la para ver se a configuração teve efeito (talvez seja necessário realizar um hard refresh).

Agora vamos ao site do bootstrap ver que temos uma gama enorme de configurações das fontes, mas a configuração que queremos ver, corresponde à este link: https://getbootstrap.com/docs/5.3/utilities/text/.
Se quisermos alterar o tamanho da fonte por exemplo, vamos à parte font size, lá veremos os códigos para realizar isso, então voltamos ao nosso arquivo navbar.html na mesma parte que alteramos anteriormente e logo após logo vamos adicionar fs-4 (lembrando que quanto menor o número, maior será a fonte), ficando assim:

```
      <a class="navbar-brand logo fs-3"
href="#">Navbar</a>
```

Vamos dar um hard refresh e vamos e se tudo der certo, veremos que a fonte da logo aumentou.

Outra alteração que vamos fazer é no arquivo main.css, como fontes de tamanho maior podem interferir no espaçamento, vamos adicionar uma configuração para mitigar isso, ficando assim:

```
.logo{
    font-family: 'Archivo', sans-serif;
    letter-spacing: 0.5px;
}
```

Agora, para o texto principal, no google fonts vamos utilizar a fonte chamada Barlow e vamos selecionar as seguintes opções: Regular 400, Medium 500, SemiBold 600, Bold 700, Black 900

Vamos construir uma ‘entrada’ chamada *test* para salvar o CSS rules que vamos utilizar posteriormente, vamos fazer assim para não termos que voltar ao site do google fonts apenas para isso. O processo é o mesmo da etapa anterior.

E para o texto do conteúdo do blog, vamos utilizar a fonte Merriweather e vamos selecionar a seguinte opção: Regular 400, e vamos adicionar o CSS rules no na entrada *test* criada anteriormente.

Construindo componentes de Bootstrap

Vamos fazer um componente específico para o app *blog*, então vamos na pasta *templates* > *blog* e vamos criar uma nova pasta chamada *components* e dentro desta pasta vamos criar um novo arquivo chamado *splash.html*.

"página de apresentação" (ou "splash page" em inglês) é uma tela de introdução ou página inicial que os usuários veem quando visitam um site pela primeira vez.

Vamos fazer uma recapitulação do que fizemos até agora: a nossa navbar é está contida em um 'container' que representa uma seção da nossa página da web. E essa seção (container) é especificada com uma quantidade máxima de largura.
Então se quisermos construir outro container embaixo ou em cima, queremos que este container esteja no tamanho correto.
Se observarmos o arquivo *navbar.html* dentro da pasta base, veremos que a nossa navbar está dentro de um container, o tamanho não está explicitamente especificado, já que pegamos diretamente do bootstrap pode estar sendo determinado pelo css ou alguma outra questão.

Vamos agora construir o nosso novo container Splash, vamos ao arquivo *splash.html* que criamos e vamos colocar o seguinte código:

```
<div class ="container">

    conteúdo aqui
</div>
```

Agora vamos no nosso arquivo *index.html* e vamos incluir este arquivo lá, fazemos isso com o seguinte código logo abaixo da navbar:

```
{% include "./components/splash.html" %}
```

Agora vamos adicionar uma linha com borda para este novo container. Há duas formas de fazer, a primeira é assim:

```
<div class ="container border-bottom border-2 border-
dark">
```

```
    conteúdo aqui
</div>
```

Definimos um container, a posição da borda, sua largura e sua cor.

Mas se olharmos na página com a aplicação em execução, veremos que a borda tem um ‘limite’, queremos que ela cubra a linha inteira.

Para ficar da forma que queremos, precisamos adaptar o código para ficar assim:

```
<div class="border-bottom border-2 border-dark">
    <div class="container">

        conteúdo aqui
    </div>
</div>
```

O que fizemos aqui, foi criar outra div e mover o conteúdo do container da div anterior para ela e então colocar a div anterior dentro dessa nova div, como se fosse um container dentro do container.
Agora ao olhar a página da nossa publicação veremos a linha inteira.

Vamos agora adicionar alguns textos (conteúdo), para isso vamos usar as tags h1 <h1>, essas tags vão do h1 até h6, o ponto principal da utilização dessas tags são para ajudar a guiar o navegador. Por exemplo: se estivermos utilizando algum reader irá diretamente ao h1 primeiro porque são os headings (cabeçalhos/títulos) da página, para motores de busca e otimização pode dizer ao google e outros motores de busca qual é a coisa mais importante nessa página e isso dá contexto à página.
Então a tag <h1> é a mais importante das tags, e elas vão até o <h6>

Vamos utilizar a <h1> para criar um sistema de prioridade para o conteúdo da nossa página. Além disso, vamos adicionar também uma classe dentro da tag h1 e vamos utilizar o bootstrap para definir o tamanho da fonte, e vamos chamar de *Byte Blog.*, o código final ficará assim:

```
<div class="border-bottom border-2 border-dark">
    <div class="container">

        <h1 class="display=1">Byte Blog.</h1>

    </div>
</div>
```

Lembra das fontes que separamos que estão na pasta *static* > *css* e no arquivo *main.css* para utilizar depois? Vamos utilizar a Barlow agora, vamos retirar da ‘entrada’ test que criamos e criar uma nova ‘entrada, o código ficará assim:

```
html,
body {
    font-family: 'Barlow', sans-serif;
}
```

E de volta ao arquivo *splash.html*, vamos adicionar um pouco de texto. em baixo da tag h1, vamos abrir uma tag de parágrafo <p>, e vamos escrever o seguinte texto: “Atualizações sobre nossos projetos, eventos, cursos e assuntos de tecnologia relevantes.” o código ficará assim:

```
<div class="border-bottom border-2 border-dark">
    <div class="container">

        <h1 class="display=1">Byte Blog.</h1>
        <p>
            Atualizações sobre nossos projetos, eventos,
            cursos e assuntos de tecnologia relevantes.
        </p>
    </div>
</div>
```

Agora vamos estilizar o novo texto, para isso vamos iniciar uma nova classe dentro da tag <p>, adicionando o seguinte conteúdo:

```
        <p class="fs-4 lh-1 text-muted">
```

Definimos o tamanho da fonte para 4, a line height para deixar mais curta ou maior e vamos adicionar um pouco de cor com o text-muted, é uma classe embutida no bootstrap o que vai deixar o texto meio cinzento para gerar o contraste entre o texto e o cabeçalho.

E agora vamos deixar a fonte um tanto quanto negrito, mas vamos fazer isso no nosso arquivo *main.css* e vamos fazer uma nova entrada chamada ficando assim:

```
.fw-500 {
    font-weight: 500;
}
```

E vamos adicionar essa nova configuração no nosso arquivo *splash.htm*, ficando assim:

```
        <p class="fs-4 lh-1 text-muted fw-500">
```

Ao mudar a dimensão da página, vemos que o texto tenta sempre se centralizar, mas não queremos isso. Entrão para corrigir isso vamos adicionar algumas novas colunas, no bootstrap existe algo chamado sistema grid (https://getbootstrap.com/docs/5.3/layout/grid/), nós vamos utilizar este sistema grid pois ele trás uma série de containers e colunas para

alinhar o conteúdo do layout. Este sistema grid é feito com o flexbox. Ou seja, vamos construir blocos para criar layouts para que possamos adicionar conteúdo nestes espaços. Vamos pegar o primeiro exemplo do sistema grid do bootstrap, vamos deixar apenas duas colunas e remover o container text-center, adicionar esse sistema dentro do escopo do container que já temos criado e posteriormente adicionar nosso conteúdo dentro da primeira coluna, ficando assim:

```
<div class="border-bottom border-2 border-dark">
    <div class="container">
        <div class="row">
            <div class="col">
                <h1 class="display=1">Byte Blog.</h1>
                <p class="fs-4 lh-1 text-muted fw-500">
                    Atualizações sobre nossos projetos,
eventos, cursos e assuntos de tecnologia relevantes.
                </p>
            </div>
            <div class="col">
                Column
            </div>
        </div>
    </div>
</div>
```

Se olharmos a nossa página agora, veremos o conteúdo que adicionamos perfeitamente alinhado com a primeira e segunda coluna (independente da segunda ter ou não conteúdo).

Ainda dentro da coluna, abaixo do paragrafo que criamos com o nosso conteúdo, vamos adicionar um botão (https://getbootstrap.com/docs/5.3/components/buttons/) a versão dark do primeiro exemplo.
Vamos fazer algumas alterações para suavizar as bordas e mudar o tamanho da fonte de escrita do botão, ficando assim:

```
<button type="button" class="btn btn-dark rounded-pill
fw-500 fs-5">Comece a leitura</button>
```

Vamos agora adicionar padding (preenchimento) do botão (podemos pesquisa no google para ver como funciona as regras do bootstrap) e do de parte do conteúdo, o código ficará assim:

```
<div class="col py-1 py-md-5">
```

Aqui estamos especificando que em telas pequenas o padding ficará em 1 (quanto maior o número, maior será o padding), e em telas de tamanho mais largo o padding ficará em 5.

```
<button type="button" class="btn btn-dark rounded-pill
fw-500 fs-5 px-4">Comece a Leitura</button>
```

E aqui no botão, fizemos algo similar também adicionando o padding após o definirmos o tamanho da fonte (fs-5).

Vamos adicionar padding entre o texto e o botão, fazemos isso da seguinte forma:

```
<p class="fs-4 lh-1 text-muted fw-500 pb-4">
```

O código final atualizado ficará assim:

```
<div class="border-bottom border-2 border-dark">
    <div class="container">
        <div class="row">
            <div class="col py-1 py-md-5">
                <h1 class="display=1">Byte Blog.</h1>
                <p class="fs-4 lh-1 text-muted fw-500 pb-
4">
                    Atualizações sobre nossos projetos,
eventos, cursos e assuntos de tecnologia relevantes.
                </p>
                <button type="button" class="btn btn-dark
rounded-pill fw-500 fs-5 px-4">Comece a leitura</button>
            </div>
            <div class="col">
            </div>
        </div>
    </div>
</div>
```

A estratégia de utilizar duas colunas para alinhar o nosso conteúdo foi apenas por questões didáticas, há outras formas de abordar esse problema. Primeiro, vamos apagar a coluna secundária que não estamos utilizando, se formos ver a nossa página, a coluna remanescente irá utilizar todo o espaço novamente.
O que podemos fazer é especificar o tamanho da coluna para 6 blocos, ficando assim:

```
<div class="col-6 py-1 py-md-5">
```

O tamanho máximo das colunas é de 12 blocos, mas como especificamos 6 blocos, a coluna irá utilizar apenas metade da largura disponível, sem a necessidade de duas colunas.

Mas quando estivermos em uma tela pequena, queremos que essa coluna preencha todos 12 blocos, fazemos isso adicionando mudando o col-6 para col-12, adicionamos col-md-6 para definir o tamanho de telas médias, vamos centralizar o texto se a tela for pequena e alinhar para a esquerda(start) se for média ou grande, o código ficará assim:

```
<div class="col-12 col-md-6 py-1 py-md-5 text-center
text-md-start">
```

Renderizando dados para o template

Nós vamos demonstrar todos os posts do banco de dados exibidos na página principal e nós vamos permitir ao usuário fazer o 'Lazy loading' do conteúdo à medida que vai rolando a página para baixo dependendo da quantidade de publicações que tivermos na nossa base de dados.
Nós vamos utilizar html para paginar e para carregar os dados à medida que o usuário rolar para baixo a página.

Nós vamos ter que construir um novo template, mas antes, no nosso arquivo *index.html* que está no path *templates* > *blog*, vamos adicionar logo abaixo do include do arquivo *splash.html* o um div com um container, ficando assim:

```
<div class="container">
    <div class="row">
        <div class="col-12 col-lg-8">a</div>
        <div class="d-none d-lg-block col-4">b</div>
    </div>
</div>
```

Aqui nós criamos um container e usamos a classe row para criar um contêiner que envolve as colunas em um layout de grade. O principal propósito da classe row é criar uma linha na qual as colunas serão organizadas.

Em seguida definimos a classe col-12, que indica que esta coluna deve ocupar 12 espaços (ou 100%) na grade em telas de todos os tamanhos. Isso significa que esta coluna ocupará a largura total da tela em qualquer dispositivo, a classe col-lg-8 indica que, em telas de largura média ou grande (a partir do breakpoint definido para "large", geralmente 992 pixels), a coluna deve ocupar 8 espaços na grade, deixando 4 espaços vazios para outras colunas. Isso significa que, em telas grandes, a coluna ocupará 8/12 da largura da tela.

Em seguida definimos a classe d-none que indica que esta coluna estará oculta em todas as telas (dispositivos) de tamanho pequeno ou maior. A classe d-lg-block indica que a coluna ficará visível em telas de tamanho grande (a partir do breakpoint definido para "large"). Portanto, ela ficará visível apenas em telas maiores, mas estará oculta em telas menores. A classe col-4 indica que, quando visível, a coluna ocupará 4 espaços (ou 1/3) na grade em telas grandes. Isso significa que, em telas grandes, a coluna ocupará 4/12 da largura da tela.

Vamos agora pegar os dados que queremos exibir, vamos ir na pasta da nossa aplicação *blog* (então vamos para a pasta chamada *blog*) e vamos abrir o arquivo *views.py*.
Note que já criamos a view para a nossa home e especificamos que queremos coletar e exibir nossos dados em forma de lista (estamos usando o *ListView*) e isso nos dará a ferramenta que precisamos para selecionar a base de dados e teremos um query acontecendo no 'back-end'.

E agora vamos fazer algumas alterações. Vamos adicionar uma referência dos dados que estão sendo coletados dessa classe (ou dessa view) e vamos utilizar essa referência no nosso template para que possamos acessar os nossos dados.
O código atualizado da *view.py* ficará assim:

```
class HomeView(ListView):
    model = Post
    template_name = "blog/index.html"
    context_object_name = "posts"
```

Nós nomeamos essa referência como "posts".

Agora vamos voltar ao arquivo *index.html* que está localizado no path *templates* > *blog* e vamos realizar um for loop onde estava a coluna com o conteúdo "a" que criamos, a alteração ficará assim:

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

O que fizemos aqui foi o seguinte, relembrando o arquivo *views.py*, para cada post que temos na nossa base de dados, nós vamos adicionar eles no ***context_object_name*** que chamamos de posts, e então demonstrar o título da publicação feita.

O código completo do *index.html* está assim:

```
{% extends "../base/base.html" %}
{% block title %}Index Page{% endblock %}

{% block template %}
{% include "../base/navbar.html" %}
{% include "./components/splash.html" %}

<div class="container">
```

```
    <div class="row">
        <div class="col-12 col-lg-8">

            {% for post in posts %}
            <p>
            {{post.titulo}}
            </p>
            {% endfor %}

        </div>
        <div class="d-none d-lg-block col-4">b</div>
    </div>
</div>


{% endblock %}
```

Se iniciarmos a nossa aplicação agora, veremos o título das publicações que foram feitas.

Agora que já sabemos como extrair dados da nossa base de dados, precisamos transformar esses títulos em links e estilizar sua exibição.
Vamos começar transformando a tag <p> em uma tag <a>, pois é esta tag que é usada para criar hiperlinks (links) em páginas da web e adicionar uma classe vazia, ficando assim:

```
{% for post in posts %}
<a class="">
{{post.titulo}}
</a>
{% endfor %}
```

Se atualizarmos agora veremos que os títulos são dois links.

Um link vai precisar para um endereço para especificar seu caminho, fazemos isso adicionando antes da classe um href, nós vamos ter que implementar uma forma de acessar cada uma das publicações através dos links de seus respectivos títulos, mas faremos isso posteriormente, até o momento o código está assim:

```
{% for post in posts %}
<a href="" class="">
{{post.titulo}}
</a>
{% endfor %}
```

Vamos estilizar um pouco mais, colocando o bloco de post.titulo dentro de uma div, e dentro dessa div, nós vamos construir um card, se formos na documentação do bootstrap (https://getbootstrap.com/docs/5.3/components/card/) veremos que um card é uma espécie comum de container(elemento) que vamos encontrar em qualquer website.
Os cards nos permite adicionar diferentes seções como podemos ver os modelos na documentação do bootstrap, e isso será algo que vamos utilizar para construir uma seção para cada um das nossas publicações que vamos exibir da base de dados.

E para construir esse card, vamos adicionar uma classe card dentro da div que adicionamos, ficando assim:

```
<div class="card ">
    {{post.titulo}}
</div>
```

Se atualizarmos a nossa página, veremos que já há um protótipo de card lá com uma borda onde o card começa e termina.

Nós queremos remover essa borda, podemos fazer isso apenas adicionando ***'border-0'***, mas inicialmente é bom deixar para termos uma noção do tamanho do espaço que estamos trabalhando.
Mas por agora vamos remover essa borda e adicionar um padding na parte inferior, deixando o código assim:

```
<div class="card border-0 pb-3 pb-sm-5">
    {{post.titulo}}
</div>
```

Temos um padding entre as duas publicações, o primeiro pb-3 define o padding para telas de tamanho normal, a segunda pb-sm-5 define o padding na parte inferior em telas pequenas.

Vamos agora criar outra div definida como um row, que terá duas células, a primeira terá o nosso título e a segunda terá a imagem que irá definir a nossa publicação (colocamos o espaço da imagem com a letra ***a***, o código está assim:

```
<a href="" class="">
    <div class="card border-0 pb-3 pb-sm-5">
        <div class="row">
            <div class="col-8">{{post.titulo}}</div>
            <div class="col-4">a</div>
        </div>
    </div>
</a>
```

Se atualizarmos a nossa aplicação agora, veremos o título à esquerda e o local onde terá a imagem, representado pela letra ***a*** na direita.

Vamos agora adicionar junto ao nosso título, o autor, a data de criação e o subtítulo. Mas antes vamos remover a cor azul e o sublinhado dos links das publicações, fazemos isso da seguinte forma:

```
<a href="" class="text-decoration-none">
```

E então vamos estilizar essa coluna para adicionar as demais informações da nossa publicação, para isso vamos utilizar o bloco onde está o nosso título, vamos apagar o seu conteúdo e abrir uma nova tag com um card body e especificar o padding para zero, vamos adicionar o nosso ***título*** com uma tag <h1> com uma classe especificando a margem inferior igual a um, depois uma tag <h2> para o ***subtítulo***, então vamos definir a classe como text-muted, e depois vamos adicionar o ***autor***, com uma div recebendo uma classe pb-2 text-body e por fim a seção da ***data de publicação*** que será uma div com uma classe recebendo text-muted e definindo o formato da exibição da data apenas para o mês e o dia.
O código completo ficará assim:

```
<a href="" class="text-decoration-none">
    <div class="card border-0 pb-3 pb-sm-5">
        <div class="row">
            <div class="col-8">
                <h1 class="mb-1">{{post.titulo}}</h1>
                <h2 class="text-
muted">{{post.subtitulo}}</h2>
                <div class="pb-2 text-
body">{{post.autor}}</div>
                <div class="text-muted">{{post.criado_em|
date:"M d"}}</div>
            </div>
            <div class="col-4">a</div>
        </div>
    </div>
</a>
```

Agora vamos definir o local onde vamos armazenar a imagem que vamos utilizar para especificar as nossas publicações.
Vamos criar uma pasta chamada *imagens* dentro da pasta *static*, e vamos colocar o arquivo *LOGO.png* dentro dessa pasta.
E no nosso arquivo *index.html*, abaixo do extends, vamos colocar o seguinte código:

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

Quanto a coluna da imagem, o código ficará assim:

```
<div class="col-4">
    <img src="{% static 'imagens/LOGO.png' %}"
class="img-fluid float-end">
</div>
```

Vamos adicionar um padding global no container inteiro com pt-5, o código ficará assim:

```
<div class="container pt-5">
```

Vamos alterar o tamanho da fonte do subtítulo, ficando assim:

```
<h2 class="text-muted fs-6">{{post.subtitulo}}</h2>
```

Vamos também alterar a fonte do título, a sua cor e deixar um pouco mais em negrito, ficando assim:

```
<h1 class="mb-1 fs-4 text-body">{{post.titulo}}</h1>
```

Vamos configurar o escalonamento automático dependendo do tamanho da tela que estiver vendo o nosso site, primeiro passo que vamos fazer é é remover tamanho da fonte do título, deixando assim:

```
<h1 class="mb-1 text-body fw-bold">{{post.titulo}}</h1>
```

A segunda coisa que vamos fazer é criar uma nova entrada no arquivo *main.css* que está no path *static* > *css*, ficando assim:

```
.infscroll h1 {
    font-size: 22px;
    letter-spacing: -0.6px;
}

@media only screen and (max-width: 768px) {
    .infscroll h1 {
```

```
        font-size: 15px;
        letter-spacing: 0.0px;
    }
}
```

A primeira interação vai definir o padrão para telas grandes, e a segunda é para telas com resolução inferior a 768px (podemos verificar isso no breakpoints do bootstrap)

Agora vamos definir essa entrada como uma classe no arquivo *index.html*, na parte logo abaixo do for, ficando assim:

```
<a href="" class="text-decoration-none infscroll">
```

E agora vamos fazer o subtítulo sumir caso a tela seja de tamanho pequeno, fazemos isso adicionando o d-none e o d-md-block após o fs-6, o código ficará assim:

```
<h2 class="text-muted fs-6 d-none d-md-
block">{{post.subtitulo}}</h2>
```

Agora vamos aplicar o autoescalonamento do tamanho da fonte para o autor e para a data também, vamos aproveitar a entrada que criamos no arquivo *main.css* e vamos colocar mais uma condição a ela, ficando assim:

```
@media only screen and (max-width: 768px) {
    .infscroll h1 {
        font-size: 15px;
        letter-spacing: 0.0px;
    }

    .infscroll-autor,
    .infscroll-data {
        font-size: 13px;
    }
}
```

E no arquivo *index.html* vamos atualizar o autor e a data deixando cada um assim:

```
<div class="pb-2 text-body infscroll-autor">Autor:
{{post.autor}}</div>
<div class="text-muted infscroll-data">{{post.criado_em|
date:"M d"}}</div>
```

Gerando dados com Factory Boy e Faker

Nós vamos gerar publicações para testar as próximas configurações no front-end que vamos utilizar.

Para gerar essas publicações, vamos para a nossa pasta *blog* que está dentro da pasta *ByteBuilders*, dentro da pasta *blog* vamos criar uma novo arquivo chamado *factory.py*, e dentro deste arquivo nós vamos utilizar o factory boy(factories) e o faker para gerar esses dados, vamos começar vamos verificar se temos o factory boy instalado, vamos utilizar o comando: ***pip freeze*** no terminal, se não tiver o factory boy instalado, vamos instalá lo com o seguinte comando: ***pip install factory-boy***.

Vamos abrir o arquivo models.py ao lado do arquivo factory.py no vscode, já que vamos utilizar elementos que estão contidos ali.

Para começar, vamos importar o factory, como vamos fazer publicações será necessário especificar um usuário, já que cada publicação tem um autor, então precisamos importar os usuários também e devemos importar o nosso modelo de dados, que chamamos de *Post* e por fim vamos importar o faker, fazemos isso da seguinte forma:

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

from .models import Post
```

Agora vamos criar uma classe chamada ***PostFactory***, factory tem ferramentas específicas para o Django, e nós vamos passar como parâmetro dessa classe uma dessas ferramentas, o ***DjangoModelFactory***, ficando assim:

```
class PostFactory(factory.django.DjangoModelFactory):
```

Agora vamos criar uma classe aninhada, chamada Meta, e especificar o modelo (table) que vamos utilizar, ficando assim:

```
    class Meta:
        model = Post
```

A classe aninhada *Meta* é usada para definir metadados específicos para esta fábrica. No caso, você está especificando o modelo ao qual essa fábrica está associada. Isso significa que, quando você usa esta fábrica para criar objetos, ela sabe que deve criar instâncias do modelo "Post."
Agora vamos definir o nosso modelo adicionando alguns dos dados que queremos inserir nesse modelo (tabela).
Vamos adicionar alguns dados do *models.py*, então vamos começar adicionando o titulo e subtitulo com o faker (o faker irá gerar dados aleatórios para nós como nomes, datas, entre outros… podemos ver mais detalhes na documentação do faker), no campo do autor, vamos definir para utilizar um autor com username igual a filipe ou criar um, ficando assim:

```
class PostFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Post

    titulo = factory.Faker("sentence", nb_words=12)
```

```
    subtitulo = factory.Faker("sentence", nb_words=12)
    slug = factory.Faker("slug")
    autor = User.objects.get_or_create(username="filipe")
[0]
```

O ***criado_em*** e ***atualizado_em*** será criado automaticamente, por isso não especificamos o nosso modelo.

Vamos agora utilizar o decorador lazy_attribute e vamos criar uma função para gerar valor para o nosso campo de texto (conteudo), e outras funções para gerar automaticamente conteúdo fictício de post composto por cinco parágrafos quando uma instância da fábrica for criada. A geração do conteúdo é preguiçosa, ou seja, só acontece quando o atributo é acessado, economizando recursos de processamento se o atributo não for utilizado. Mas antes devemos criar uma instância da classe faker para utilizá-lá.
Fazemos isso assim:

```
FAKE = faker.Faker()
```

E para utilizar o decorador lazy_attribute fazemos assim:

```
    @factory.lazy_attribute
    def conteudo(self):
        publi = ""
        for _ in range(0, 5):
            publi += "\n" +
FAKE.paragraph(nb_sentences=30) + "\n"
        return publi
```

@factory.lazy_attribute é um decorador usado em conjunto com o Factory Boy para criar atributos preguiçosos (lazy attributes). Isso significa que o valor do atributo não será gerado imediatamente ao criar a instância da fábrica, mas sim quando o atributo for acessado pela primeira vez.
def conteudo(self): é a definição da função que será usada para gerar o valor do atributo "conteudo". Ela recebe o parâmetro self, que se refere à instância da fábrica em que esse atributo será usado.
Dentro da função *conteudo*, é criada uma variável chamada *publi* inicializada como uma string vazia.
Em seguida, um loop for é utilizado para concatenar texto à variável *publi*. O loop roda 5 vezes (de 0 a 4).
Dentro de cada iteração do loop, uma nova string é gerada usando a função FAKE.paragraph(nb_sentences=30). Isso cria um parágrafo de texto fictício com 30 frases, simulando o conteúdo de um post.
O parágrafo gerado é então concatenado à variável publi após um caractere de quebra de linha ("\n"). Isso ajuda a formatar o texto, adicionando uma linha em branco entre os parágrafos.
Após o loop, a variável publi contém cinco parágrafos fictícios.
Finalmente, a função retorna a string *publi*, que será o valor do atributo "conteudo" quando a instância da fábrica for criada.

como definimos isso

E por fim vamos definir o status da publicação como publicado, ficando assim:

```
    status = "published"
```

O código completo ficará assim:

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

from factory.faker import faker

FAKE = faker.Faker()


class PostFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Post

    titulo = factory.Faker("sentence", nb_words=12)
    subtitulo = factory.Faker("sentence", nb_words=12)
    slug = factory.Faker("slug")
    autor = User.objects.get_or_create(username="filipe")
[0]

    @factory.lazy_attribute
    def conteudo(self):
        publi = ""
        for _ in range(0, 5):
            publi += "\n" +
FAKE.paragraph(nb_sentences=30) + "\n"
        return publi

    status = "published"
```

Vamos precisar agora acessar o shell da nossa aplicação para executar isso, fazemos isso da seguinte forma, com o terminal estando no diretório do arquivo *manage.py*, vamos

digitar: ***./manage.py shell*** e em sequência vamos digitar o seguinte comando: ***from ByteBuilders.blog.factory import import PostFactory***, depois vamos digitar: **public = PostFactory.create_batch(100)** estamos especificando nesse último comando para fazer 100 publicações para nós e então podemos sair do console digitando ***exit()***.

Se executarmos o projeto agora, veremos cerca de 100 publicações mais as que já existiam.

Paginação de postagem HTMX

HTMX é uma biblioteca JavaScript que permite atualizar partes específicas de uma página da web de forma dinâmica, sem a necessidade de recarregar a página inteira.
Isso é feito usando solicitações AJAX para buscar dados do servidor e atualizar o conteúdo da página em tempo real.
HTMX é uma alternativa ao uso tradicional de frameworks front-end como o React ou o Vue.js, pois é mais leve e se concentra em melhorar a interatividade de páginas já existentes.

Imagine se tivermos umas vinte mil publicações ou mais no nosso site? Não podemos carregar isso tudo de uma vez, por questões óbvias como consumo excessivo de recursos sendo uma das principais preocupações.

Então vamos configurar para quebrar esse fluxo de publicações em pedaços menores e então carregá-los à medida que forem requisitados, isso irá aperfeiçoar a performance da nossa página.

Nós vamos utilizar o HTMX para carregar (paginar) a seção de publicações do nosso site para que sejam carregadas apenas 10 publicações por vez.
Quando o usuário for para a página 10 das publicações serão carregadas e a medida que ele for rolando para baixo, a aplicação enviará um requerimento para o banco de dados e para o nosso servidor para o Django e então mais 10 serão carregadas, isso vai acontecendo de 10 em 10 de cada vez até o fim das publicações.

E como mencionado anteriormente, nós faremos isso utilizando o AJAX para enviar um requerimento para o servidor sem a necessidade de ter que recarregar a página.

Primeiro de tudo, vamos criar uma nova pasta dentro da pasta *static* chamada *js*, dentro desta nova pasta vamos criar um arquivo chamado *htmx.min.js*. Este arquivo que irá conter todo o javascript necessário para a nossa paginação, mais detalhes na documentação do htmx: (https://htmx.org/docs/). No site da documentação vamos fazer um download ou copiar o código que deve conter no arquivo *htmx.min.js*. Nós vamos utilizar o trigger (gatilho) chamado revelead do htmx.

Deve incluir o arquivo *htmx.min.js* no arquivo *base.html* que está no path *ByteBuilders* > *templates* > *base*. Vamos adicionar o seguinte script na parte final dentro da tag head:

```
<script src="{% static 'js/htmx.min.js' %}"
defer></script>
```

Agora, no nosso projeto, vamos ao seguinte path *ByteBuilders* > *templates* > *blog* > *components* e vamos criar um arquivo chamado *post-list-elements.html*, este será mais um componente separado que irá carregar toda as publicações e posteriormente poderemos utilizar esse componente novamente para em seguida, para gerar a saída dos novos dados que foram retornados para quando o usuário rolar para baixo.

Vamos remover todo o loop do arquivo *index.html* e vamos adicionar no *post-list-elements.html* (não esqueça de levar o ***{% load static %}*** também, no *index.html* vamos incluir o novo template no local onde o código anterior estava, ficando assim o container:

```
<div class="container pt-5">
    <div class="row">
        <div class="col-12 col-lg-8">

            {% include "./components/post-list-
elements.html" %}

        </div>
        <div class="d-none d-lg-block col-4">b</div>
    </div>
</div>
```

Indo agora para o arquivo *views.py* que está na pasta *ByteBuilders > blog*. Neste arquivo podemos ver a única view que nós temos, que é a que utilizamos para construir e exibir a nossa home page.
Vamos utilizar aqui algumas características de paginação do Django que irá nos permitir dividir os nossos dados em páginas múltiplas, mas não vamos utilizar aquele contexto de enumeração de páginas (clicar em uma página para ir pra outra página… etc), e como já mencionamos nós vamos utilizar o AJAX para enviar o requerimentos e pegar os dados (da próxima página de dados) e retornar diretamente para a tela.

Nós configuramos a *view.py* fazendo uma especificação da seguinte forma após o ***context_object_name***:

```
    paginate_by = 10
```

Agora ao chegar no final da nossa página, ainda não carregará as próximas 10 publicações, vamos precisar utilizar o htmx para adicionar o trigger(gatilho) e o revelead do htmx, vamos ir ao arquivo *post-list-elements.html* e vamos adicionar um novo componente:

```
{% load static %}
{% for post in posts %}
<a href="" class="text-decoration-none infscroll">
```

```
    {% if forloop.last %}
    <div class="card border-0 pb-4 pb-sm-5" hx-
trigger="revealed">
        {% else %}
        <div class="card border-0 pb-4 pb-sm-5">
            {% endif %}
            <div class="row">
```

Mudamos um pouco a estrutura do que tínhamos antes, colocamos condições no nosso loop adicionando o trigger, portanto sempre no último elemento da página, nós teremos o trigger que será ativado uma vez quando um elemento é rolado pela primeira vez para a janela de visualização (viewport).
Então o que fizemos nessa alteração aqui é que sempre quando atingirmos o último item do loop, o trigger (gatilho) será ativado.

```
{% if forloop.last %}
<div class="card border-0 pb-4 pb-sm-5" hx-
trigger="revealed">
```

Nessa parte em específico nós estamos informando, se for o último post vamos utilizar essa div, que é a que possui o gatilho. Se não vai continuar usando a div normal, que é essa daqui:

```
{% else %}
<div class="card border-0 pb-4 pb-sm-5">
```

Se observarmos agora na página, notamos que houve uma perda na formatação, vamos corrigir isso com uma simples configuração, mas antes vamos inspecionar o elemento, pois, veremos que a última publicação terá o hx-trigger, e as demais não terão, isso significa que o nosso loop e sua condição estão funcionando.

Agora para corrigir a formatação, vamos remover o infscroll dessa parte:

```
<a href="" class="text-decoration-none infscroll">
```

E vamos adicioná-lo no título dentro da div no h1, ficando assim:

```
<div class="row">
    <div class="col-8">
```

```
        <h1 class="mb-1 text-body fw-bold infscroll-
titulo">{{post.titulo}}</h1>
```

Após essa alteração é necessário atualizar o arquivo *main.css* que está no seguinte path *static* > *css*. A alteração será aqui:

```
.infscroll-h1 {
    font-size: 22px;
    letter-spacing: -0.6px;
}

@media only screen and (max-width: 768px) {
    .infscroll-h1 {
        font-size: 15px;
        letter-spacing: 0.0px;
    }
```

Vamos substituir os h1 por título, deixando assim:

```
.infscroll-titulo {
    font-size: 22px;
    letter-spacing: -0.6px;
}

@media only screen and (max-width: 768px) {
    .infscroll-titulo {
        font-size: 15px;
        letter-spacing: 0.0px;
    }
```

Vamos criar um AJAX get requerimento. Vamos fazer isso adicionando um ***hx-get*** após o ***revealed***,hx-get é usado para definir o URL ou a rota que deve ser solicitada quando o evento especificado na hx-trigger ocorrer uma ação, vamos implementar isso da seguinte forma:

```
<div class="card border-0 pb-4 pb-sm-5" hx-
trigger="revealed" hx-get="{% url 'homepage' %}">
```

Isso será aplicado APENAS no último elemento. Se atualizarmos a página, veremos que há esse requerimento e a homepage é expressada com uma simples barra /, isso irá enviar um requerimento AJAX para o servidor para pegar mais dados.

O que precisamos fazer agora é configurar a nossa view para detectar que um requerimento foi enviado pela nossa homepage e então fornecer ou enviar as próximas 10 publicações para a homepage.
Para fazer isso, vamos ter que instalar o pacote: ***pip install django-htmx***(https://pypi.org/project/django-htmx/).
Esse pacote irá nos fornecer alguns recursos diferentes que irá nos permitir capturar os requerimentos htmx e então poderemos realizar alguma ação em cima disso.

Após a instalação devemos registrar o django htmx no base.py arquivo que está dentro da pasta settings.

```
INSTALLED_APPS = [
    'django_htmx',
]

MIDDLEWARE = [
    'django_htmx.middleware.HtmxMiddleware',
]
```

Agora nós vamos enviar o template de volta para a homepage, vamos fazer isso criando uma função no arquivo *view.py* que está na pasta *ByteBuilders* > *blog*.
Isso é necessário porque sempre que precisamos de mais dados, nós precisamos de um template para mostrar esses dados, e o que vamos fazer é sobrescrever o *get_template_names* e sempre que um requerimento htmx for feito, nós podemos enviar apenas o *post-list-elements.html* template, pois, este é o template que nós vamos realizar o loop por todos as publicações e então produzir a saída dos dados(exibir os dados) que estão sendo requeridos.

Então vamos adicionar a seguinte função no nosso arquivo *view.py*:

```
    def get_template_names(self):
        if self.request.htmx:
```

Sempre que enviarmos uma requisição htmx, no back-end (durante o requerimento) há alguns metadados que pode detectar, neste caso a informação do htmx e isso significa que a nossa homepage ativou o código htmx que está na última publicação(item) renderizado, então isso significa que podemos especificar para o htmx realizar algum tipo de tarefa.

Vamos pedir para imprimir no terminal xxxxx sempre que o trigger(gatilho) for ativado.

O código completo fica assim:

```
    def get_template_names(self):
```

```
        if self.request.htmx:
            print('xxxxx')
        return "blog/index.html"
```

E como nós sobrescrevemos o método *get_template_names* nós podemos apagar a variável ***template_name***, ficando assim todo o código da *view.py*:

```
from django.shortcuts import render
from django.views.generic import ListView
from .models import Post


class HomeView(ListView):
    model = Post
    template_name = "blog/index.html"
    context_object_name = "posts"
    paginate_by = 10

    def get_template_names(self):
        if self.request.htmx:
            print('xxxxx')
        return "blog/index.html"
```

Vamos executar a aplicação e ver se está funcionando.
Se estiver vamos configurar essa função para retornar o nosso template, ficando assim o código:

```
    def get_template_names(self):
        if self.request.htmx:
            return "blog/components/post-list-
elements.html"
        return "blog/index.html"
```

Veja que definimos o paginate_by para 10 publicações de cada vez, e o que precisamos fazer quando enviamos o requerimento (request) precisamos especificar o número da página que queremos receber, por exemplo, a primeira página carregada será a página 1… após descer 10 publicações, vamos ativar o trigger (gatilho) novamente então estaremos na página 2… e assim por diante.

Para configurar isso, basta voltar ao arquivo *post-list-elements.html* e alterar essa linha:

```
<div class="card border-0 pb-4 pb-sm-5" hx-
trigger="revealed" hx-get="{% url 'homepage' %}">
```

Vamos adicionar ***?page={{ page_obj.number|add:1 }}*** estamos basicamente informando que o número da página inicial das páginas será 1 e vamos adicionar também ***hx-swap="afterend"***: Quando hx-swap é definido como "afterend", significa que o conteúdo obtido através da solicitação será inserido imediatamente após o elemento que possui o atributo hx-get. Em outras palavras, o conteúdo será adicionado logo após o fechamento da tag HTML do elemento.

O código atualizado ficará assim:

```
    <div class="card border-0 pb-3 pb-sm-5" hx-
trigger="revealed" hx-get="{% url 'homepage' %}?
page={{ page_obj.number|add:1 }}" hx-swap="afterend">
```

Tag Cloud

Há diversas ferramentas que podemos utilizar no django (https://djangopackages.org/), nós vamos utilizar o Django Taggit (https://pypi.org/project/django-taggit/). Para isso, devemos utilizar o seguinte comando no terminal: ***pip install django-taggit***, agora precisamos adicionar o taggit no *INSTALLED_APPS*, portanto devemos ir ao arquivo *base.py* que está na pasta *settings* e adicionar o ***"taggit",*** lá.

Agora vamos integrar o taggit na model, mas antes é bom relembrar que nós já criamos (migramos) a nossa base de dados, então a primeira coisa que devemos fazer é deletar a nossa base de dados e dentro da pasta *blog*, vamos apagar a pasta *migrations*.

Após realizar este passo, devemos ir para o arquivo *models.py* que está dentro da pasta *blog* e importar o TaggableManager, ficando assim:

```
from taggit.managers import TaggableManager
```

E agora vamos adicionar um novo campo para a nossa tag, ficando assim:

```
tags = TaggableManager()
```

Agora vamos recriar a nossa base de dados, e utilizar o factory que criamos anteriormente para encher a base de dados novamente, mas antes devemos adicionar as tags da nossa base de dados também no arquivo factory.py.

Nós vamos precisar adicionar as tag posteriormente quando salvarmos os dados, então vamos utilizar a ***factory.post_generation*** e criar uma função com o nome que definimos no arquivo models.py que é tags e vamos seguir a documentação passando os parâmetros lá contidos e os kwargs (argumentos de palavra-chave), ficando assim:

```
@factory.post_generation
def tags(self, create, extracted, **kwargs):
```

E então vamos criar uma condição em que caso não passem alguma tag para o nosso posts (já que elas não são obrigatórias), nós apenas retornamos, não é necessário fazer nada, ficando assim:

```
if not create:
    return
```

Mas caso alguma tag seja passada, nós precisamos extraí-las o que irá nos permitir passar as nossas próprias tags sempre quando gerarmos um novo post utilizando a factory, portanto devemos especificar na condição que se extraída (a tag for passada) devemos adicioná-la (extraída)

```
if extracted:
    self.tags.add(extracted)
```

Se não passarmos nenhuma tag, vamos adicioná-las manualmente adicionando no código

```
else:
    self.tags.add(
        "Python",
        "Django",
        "Database",
        "Pytest",
        "JavaScript",
        "VSCode",
        "Deployment",
        "Full-Stack",
        "ORM",
        "Front-End",
        "Back-End",
    )
```

O código completo ficará assim:

```
    @factory.post_generation
    def tags(self, create, extracted, **kwargs):
if not create:
    return
if extracted:
    self.tags.add(extracted)
else:
    self.tags.add(
```

```
        "Python",
        "Django",
        "Database",
        "Pytest",
        "JavaScript",
        "VSCode",
        "Deployment",
        "Full-Stack",
        "ORM",
        "Front-End",
        "Back-End",
    )
```

Agora vamos recriar a nossa base de dados, vamos recriar dentro da pasta *blog* a pasta *migrations* e dentro dela o arquivo *__init__.py* e vamos executar os seguintes comandos no terminal:
./manage.py makemigrations
./manage.py migrate

Agora vamos utilizar novamente a nossa factory para gerar mais dados, para isso vamos digitar no terminal ***./manage.py shell*** e então colar os seguintes comandos:
from ByteBuilders.blog.factory import PostFactory
publi = PostFactory.create_batch(100)
exit()

Agora vamos executar o nosso servidor novamente para ver se as publicações apareceram.

Se tudo der certo, agora vamos precisar extrair as tags e criar as tags cloud e colocá-las naquele container ‘b’ que aparece na nossa página.

Para isso devemos abrir o arquivo *index.html* que está no seguinte path *templates* > *blog*, aqui vemos o container ‘b’, vamos apagá-lo pois é aqui nós queremos extrair todas aquelas tags que criamos da nossa base de dados.
Portanto vamos ir na pasta components e criar um novo componente chamado tag-cloud que poderá ser reutilizado no resto do nosso site