# Django

## Configurações iniciais

### 1. Criação de Projetos e Aplicações

- `startproject`: Cria a estrutura básica de um novo projeto Django, incluindo diretórios e arquivos iniciais necessários.

```
django-admin startproject nome_do_projeto
```

- `startapp`: Cria a estrutura básica para uma nova aplicação dentro de um projeto Django.

```
django-admin startapp nome_da_aplicacao
```

### 2. Migrações de Banco de Dados

- `makemigrations`: Cria novos arquivos de migração com base nas mudanças detectadas nos modelos (`models`) da aplicação.

```
python manage.py makemigrations
```

- `migrate`: Aplica as migrações ao banco de dados, sincronizando o estado do banco com os modelos da aplicação.

```
python manage.py migrate
```

### 3. Interatividade e Shell

- `shell`: Abre um shell interativo Python pré-configurado com o ambiente do Django, facilitando a interação direta com os modelos e outras partes do projeto.

```
python manage.py shell
```

- `dbshell`: Abre uma interface de shell do banco de dados configurado, permitindo interagir diretamente com o banco de dados usado pelo projeto.

```
python manage.py dbshell
```

## 4. Serviço Web de Desenvolvimento

- `runserver`: Inicia o servidor de desenvolvimento do Django, permitindo que o projeto seja acessado via um navegador web para testes.

```
python manage.py runserver
```

## 5. Administração de Usuários

- `createsuperuser`: Cria um novo usuário administrador (superuser) para acessar a interface de administração do Django.

```
python manage.py createsuperuser
```

- `changepassword`: Permite alterar a senha de um usuário especificado.

```
python manage.py changepassword nome_do_usuario
```

## 6. Testes e Depuração

- `test`: Executa os testes automatizados definidos no projeto.

```
python manage.py test
```

## 7. Manutenção e Utilitários Diversos

- `collectstatic`: Coleta todos os arquivos estáticos (CSS, JavaScript, imagens) de cada aplicação para um único diretório, normalmente usado para implantação em produção.

```
python manage.py collectstatic
```

- `check`: Executa verificações de sistema e validações do projeto.

```
python manage.py check
```

## 8. Comandos Personalizados

- O Django permite que você crie comandos personalizados de gerenciamento. Isso é útil para adicionar funcionalidades específicas ao seu projeto que podem ser executadas através da linha de comando.

### Exemplos de Uso

Exemplos práticos de como você poderia usar `django-admin`:

1. **Criar um novo projeto Django:**

   ```
   django-admin startproject myproject
   ```

   Este comando cria uma nova pasta chamada `myproject` contendo a estrutura básica do projeto Django.

2. **Criar uma nova aplicação dentro de um projeto existente:**

   ```
   django-admin startapp blog
   ```

   Isso cria uma nova aplicação chamada `blog` com diretórios e arquivos padrão.

3. **Executar o servidor de desenvolvimento:**

   ```
   python manage.py runserve
   ```

   Após configurar seu projeto, esse comando inicia um servidor local acessível via `http://127.0.0.1:8000/`.

4. **Criar migrações para novas alterações em modelos:**

   ```
   python manage.py makemigrations
   python manage.py migrate
   ```

   Estes comandos geram e aplicam migrações para refletir mudanças de esquema no banco de dados.

# Criação e configuração de APPs

O comando `django-admin startapp` é usado para criar a estrutura básica de uma nova aplicação dentro de um projeto Django. Cada aplicação Django é

essencialmente um módulo que pode ter suas próprias funcionalidades, modelos de dados, visualizações, URLs, arquivos estáticos e templates.

## 1. Estrutura Criada pelo `startapp`

Quando você executa `django-admin startapp nome_da_aplicacao`, o Django cria uma pasta chamada `nome_da_aplicacao` com a seguinte estrutura de diretórios e arquivos padrão:

```
nome_da_aplicacao/
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations/
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
```

Descrição de cada um desses componentes:

- `__init__.py`: Um arquivo vazio que indica ao Python que esta pasta deve ser tratada como um pacote.
- `admin.py`: Configurações para a interface de administração do Django. Aqui você pode registrar seus modelos para que possam ser gerenciados através do site de administração do Django.
- `apps.py`: Contém a configuração da aplicação, incluindo o nome e outras configurações específicas da aplicação.
- `migrations/`: Uma pasta para armazenar os arquivos de migração de banco de dados. O arquivo `__init__.py` faz da pasta um módulo Python.
- `models.py`: Define as classes de modelo para sua aplicação, que correspondem às tabelas no banco de dados.
- `tests.py`: Contém classes de teste para a aplicação, permitindo que você escreva testes unitários e de integração.
- `views.py`: Define as funções ou classes que processam as requisições HTTP e retornam respostas.

## 2. Passos Após Criar uma Aplicação

Depois de criar uma nova aplicação, há alguns passos que normalmente são seguidos para integrá-la ao projeto:

1. **Registrar a Aplicação no Projeto**
   - Adicione o nome da aplicação à lista `INSTALLED_APPS` no arquivo de configuração `settings.py` do projeto.

     ```
     INSTALLED_APPS = [
         ...
         'nome_da_aplicacao',
     ]
     ```

2. **Definir os Modelos**
   - No arquivo `models.py`, você define os modelos de dados que a aplicação usará. Estes modelos são classes Python que herdam de `django.db.models.Model`.

     ```
     from django.db import models

     class ExemploModelo(models.Model):
         campo1 = models.CharField(max_length=100)
         campo2 = models.IntegerField()
     ```

3. **Criar e Aplicar Migrações**
   - Use os comandos `makemigrations` e `migrate` para criar e aplicar migrações com base nos modelos definidos.

     ```
     python manage.py makemigrations nome_da_aplicacao
     python manage.py migrate
     ```

4. **Configurar URLs**
   - Crie um arquivo `urls.py` na nova aplicação e defina as rotas específicas da aplicação. Em seguida, inclua estas URLs no arquivo de URLs principal do projeto (`urls.py` do projeto).

```
# urls.py na aplicação
from django.urls import path
from . import views

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

```
# urls.py no projeto
from django.urls import include, path

urlpatterns = [
    path('nome_da_aplicacao/', include('nome_da_aplic
acao.urls')),
]
```

5. **Criar Views**
   - Defina as funções ou classes em `views.py` que lidam com as requisições HTTP e retornam respostas.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, mundo! Esta é a págin
a inicial da aplicação.")
```

6. **Configurar o Admin**
   - Registre os modelos no arquivo `admin.py` para que eles sejam gerenciáveis pela interface de administração do Django.

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

admin.site.register(ExemploModelo)
```

7. **Executar comando**

- Comando completo para criar uma aplicação chamada `blog`:

```
django-admin startapp blog
```

## 3. Detalhamento Completo dos Arquivos Criados

1. `admin.py`
   - Onde você registra os modelos para serem acessíveis na interface de administração do Django.

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

admin.site.register(ExemploModelo)
```

2. `apps.py`
   - Contém a configuração da aplicação, incluindo o nome e outros parâmetros.

```
from django.apps import AppConfig

class BlogConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoFie
ld'
    name = 'blog'
```

3. `models.py`
   - Define os modelos de dados da aplicação, representando tabelas no banco de dados.

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
```

```
    created_at = models.DateTimeField(auto_now_add=Tr
u
```

4. `tests.py`
   - Contém testes para as funcionalidades da aplicação.

```
from django.test import TestCase
from .models import Post

class PostModelTest(TestCase):
    def test_string_representation(self):
        post = Post(title="My entry title")
        self.assertEqual(str(post), post.title)
```

5. `views.py`
   - Contém as funções ou classes que processam as requisições HTTP e retornam respostas.

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world! This is the bl
og index.")
```

## 4. Personalizando a Estrutura da Aplicação

Além da estrutura padrão, é comum adicionar outros diretórios e arquivos para funcionalidades específicas, como:

- `urls.py`: Para definir URLs específicas da aplicação.
- `forms.py`: Para gerenciar formulários relacionados à aplicação.
- `templates/`: Diretório para armazenar templates HTML.
- `static/`: Diretório para armazenar arquivos estáticos como CSS, JavaScript, e imagens.

## 5. Configuração de Arquivos Estáticos no Django

No Django, arquivos estáticos são aqueles que não mudam frequentemente e são servidos diretamente aos usuários. Exemplos comuns incluem arquivos CSS, JavaScript, e imagens. Para gerenciar e servir esses arquivos de maneira eficiente, o Django oferece várias configurações:

1. `STATIC_URL`: Define a URL pública para os arquivos estáticos.
2. `STATICFILES_DIRS`: Lista de diretórios adicionais onde o Django deve procurar por arquivos estáticos além das pastas 'static' de cada app.
3. `STATIC_ROOT`: O diretório onde todos os arquivos estáticos serão coletados quando o comando `collectstatic` for executado. Este é geralmente usado para preparar os arquivos para a produção.

## 6. Detalhes das Configurações

1. `STATIC_URL`

   Esta configuração define o prefixo de URL que será usado para acessar os arquivos estáticos.

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

   Por exemplo, se `STATIC_URL` for configurado como `/static/`, um arquivo CSS localizado em `static/css/style.css` será acessível via `http://seusite.com/static/css/style.css`.

2. `STATICFILES_DIRS`

   Essa configuração é uma lista de diretórios adicionais onde o Django deve procurar por arquivos estáticos, além dos diretórios 'static' presentes em cada aplicação do projeto.

   ```
   import os

   STATICFILES_DIRS = [
       os.path.join(BASE_DIR, 'setup/static'),
   ]
   ```

   - `BASE_DIR`: Geralmente definido no início do arquivo `settings.py`, representa o diretório base do projeto. Ele é configurado como o

diretório onde o `manage.py` está localizado.

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.ab
spath(__file__)))
```

- `os.path.join(BASE_DIR, 'setup/static')`: Cria um caminho absoluto para a pasta `setup/static` dentro do projeto. O Django vai procurar por arquivos estáticos também nesse diretório.

3. `STATIC_ROOT`

   O `STATIC_ROOT` define o diretório onde todos os arquivos estáticos de todas as aplicações e `STATICFILES_DIRS` serão coletados quando você executa o comando `collectstatic`.

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

   Durante o desenvolvimento, essa configuração geralmente não é usada, mas é crucial para a implantação em produção. Quando você executa o comando:

   ```
   python manage.py collectstatic
   ```

   O Django coleta todos os arquivos estáticos de cada aplicação e dos diretórios especificados em `STATICFILES_DIRS` e os copia para o diretório especificado em `STATIC_ROOT`. Isso é útil porque em um ambiente de produção, normalmente você não serve arquivos estáticos diretamente do sistema de arquivos, mas de um servidor web ou CDN (Content Delivery Network) que pode servir esses arquivos de forma mais eficiente.

**Exemplo Completo**

Para um projeto Django que tem uma pasta adicional para arquivos estáticos em `setup/static` e deseja preparar esses arquivos para produção, você pode configurar as seguintes definições no `settings.py`:

```
import os
```

```
# Diretório base do projeto
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspa
th(__file__)))

# URL que aponta para o diretório onde os arquivos estát
icos serão servidos
STATIC_URL = '/static/'

# Lista de diretórios adicionais para procurar arquivos
estáticos
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'setup/static'),
]

# Diretório onde os arquivos estáticos serão coletados p
ara produção
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

## 7. Considerações Adicionais

1. **Ambiente de Desenvolvimento vs Produção**:
   - **Desenvolvimento**: Durante o desenvolvimento, o servidor de desenvolvimento do Django serve arquivos estáticos diretamente das pastas `static` em cada app e dos diretórios listados em `STATICFILES_DIRS`.
   - **Produção**: Em produção, normalmente você usa um servidor web (como Nginx ou Apache) ou uma CDN para servir arquivos estáticos. O comando `collectstatic` coleta todos esses arquivos em `STATIC_ROOT`, de onde podem ser servidos de maneira eficiente.
2. **Uso do `collectstatic`**:
   - É importante executar `collectstatic` sempre que há mudanças em arquivos estáticos antes de implantar em produção, garantindo que todos os arquivos necessários estão disponíveis no diretório configurado em `STATIC_ROOT`.
3. **Arquivo `.gitignore`**:
   - O diretório especificado em `STATIC_ROOT` geralmente é adicionado ao `.gitignore` para evitar que os arquivos coletados sejam versionados no

controle de versão.

Exemplo de configuração `.gitignore`:

```
# Ignore the directory where static files are collected
/staticfiles/
```

### Resumo

Ao configurar `STATIC_URL`, `STATICFILES_DIRS`, e `STATIC_ROOT`, você estabelece como o Django gerencia e serve arquivos estáticos em diferentes ambientes. Essas configurações permitem que você organize seus arquivos estáticos durante o desenvolvimento e os sirva de maneira eficiente em produção.

## Boas praticas de programação

### DRY

Para aplicar o princípio DRY (Don't Repeat Yourself) em nossos templates Django, vamos criar um template base (`base.html`) que conterá as partes comuns dos outros templates, como o cabeçalho e o menu lateral. Os templates específicos (`index.html` e `imagem.html`) vão herdar desse template base e fornecerão o conteúdo específico dentro de blocos definidos. Vamos detalhar cada passo desse processo:

### 1. Passos para Criar e Usar um Template Base no Django

1. **Criar o Template `base.html`**:
   - Coloque todo o código HTML comum a ambos os templates (`index.html` e `imagem.html`) no `base.html`.
2. **Usar Herança de Template no Django**:
   - Utilize `{% extends 'galeria/base.html' %}` nos arquivos que herdarão do `base.html`.
   - Defina blocos de conteúdo que serão preenchidos pelos templates filhos (`index.html` e `imagem.html`) usando `{% block content %}{% endblock %}`.
3. **Modificar os Templates Filhos**:

- Remova o código duplicado e substitua pela extensão do template base e pelo bloco de conteúdo.

## 2. Implementação Passo a Passo

1. **Criar o Template** `base.html`
   - Na pasta `templates/galeria`, crie um arquivo chamado `base.html` e copie todo o conteúdo HTML, desde o `<body>` até o `<html>` final, dos arquivos `index.html` e `imagem.html`. Depois, adicione um bloco de conteúdo onde o conteúdo específico de cada página será inserido.

```
<!-- templates/galeria/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-widt
h, initial-scale=1.0">
    <title>Galeria</title>
    {% load static %}
    <link rel="stylesheet" href="{% static 'styles/st
yle.css' %}">
</head>
<body>
    <header>
        <h1>Alura Space</h1>
       <!-- Coloque aqui o código do cabeçalho comum,
como o logo e a barra de busca -->
    </header>
    <nav>
        <!-- Menu lateral comum -->
    </nav>
    <main>
        {% block content %}
        <!-- Conteúdo específico de cada página será
inserido aqui -->
        {% endblock %}
    </main>
    <footer>
```

```
        <!-- Rodapé comum -->
    </footer>
</body>
</html>
```

2. Modificar o `index.html` para Herança de Template
    - Agora, no arquivo `index.html`, substitua o conteúdo repetitivo pela extensão do `base.html` e use o bloco de conteúdo para adicionar o conteúdo específico da página.

```
<!-- templates/galeria/index.html -->
{% extends 'galeria/base.html' %}
{% load static %}

{% block content %}
    <div class="pagina-inicial">
        <!-- Conteúdo específico da página inicial va
i aqui -->
    </div>
{% endblock %}
```

## 3. Modificar o `imagem.html` para Herança de Template

- Faça o mesmo no `imagem.html`:

```
<!-- templates/galeria/imagem.html -->
{% extends 'galeria/base.html' %}
{% load static %}

{% block content %}
    <div class="pagina-imagem">
        <!-- Conteúdo específico da página de imagem vai
aqui -->
    </div>
{% endblock %}
```

## 4. Remover Tags Redundantes

No `index.html` e `imagem.html`, remova as tags `<body>` e `<html>` que estavam no final dos arquivos, pois essas tags já estão sendo gerenciadas no `base.html`.

**Benefícios do Template Base**

- **Manutenção Facilitada**: Qualquer alteração no cabeçalho, rodapé ou outras partes comuns precisa ser feita apenas no `base.html`, e não em cada arquivo individualmente.
- **Consistência**: Garante que todas as páginas que herdam do `base.html` mantenham uma estrutura e um estilo consistente.
- **Eficiência**: Reduz a quantidade de código duplicado e facilita a leitura e a organização do código.

## Etapas Futuras

Após estruturar os templates dessa forma, você pode continuar removendo código repetitivo do menu de navegação e do rodapé em futuras implementações, assim como mencionado. Isso pode incluir a criação de arquivos de inclusão para esses elementos e usando `{% include 'path/to/partial.html' %}` para inseri-los nos templates necessários.