

Framework Django

Augusto Hertzog

.

Contents

O que é?	4
Sobre o Tutorial.....	5
Estrutura	5
Nomenclatura das Pastas.....	5
Evitando Colisão de Nomes	5
Capturas de Tela	6
Preparando o Local de Trabalho	7
Ambiente Virtual	7
Sobre o venv	7
Criando Ambientes Virtuais	7
Ativando o Ambiente Virtual	8
Instalando os Pacotes	8
Desativando o Ambiente Virtual	8
Criando um Projeto Django.....	9
Arquivo manage.py	10
Pasta do Projeto Criada.....	10
Executando e Interrompendo o Servidor	10
Arquivo db.sqlite3.....	11
Configuração Inicial.....	12
Criando as Tabelas do Banco de Dados	12
Usuário Administrativo	13
Criando o Usuário admin	13
Acessando a Página Administrativa	13
Criando as Pastas Globais	15
Arquivo urls.py	18
Criando um Aplicativo.....	19
Comando de Criação.....	19
Registrando o Aplicativo	20
Como Funcionam as Chamadas	21
Requisição	21
Caminho da Requisição.....	21
Tags Django	21
Criando a Página Web.....	22
Entendendo os Arquivos Globais	22
Criando os Arquivos Globais	22
Arquivo HTML Global	23
Tag load	24
Tag static	24

Tag include 24

Tag block 24

O que é?

“Django é um framework para desenvolvimento rápido para web, escrito em Python, que utiliza o padrão model-template-view (MTV). Foi criado originalmente como sistema para gerenciar um site jornalístico na cidade de Lawrence, no Kansas. Tornou-se um projeto de código aberto e foi publicado sob a licença BSD em 2005. O nome Django foi inspirado no músico de jazz Django Reinhardt.

Django utiliza o princípio DRY (Don't Repeat Yourself), onde faz com que o desenvolvedor aproveite ao máximo o código já feito, evitando a repetição.”

[Wikipedia](#)

A estrutura do Django é baseada em três componentes principais :

- Model: representa a camada de dados, responsável por gerenciar informações e realizar operações de negócios;
- View: responsável por apresentar os dados ao usuário;
- Template: é a camada de apresentação, responsável por definir a estrutura básica da página, como posições de elementos e estilos;

Sobre o Tutorial

Esse é um tutorial com o passo a passo para criar um projeto e aplicativos usando o framework do Django.

Para isso, vai ser desenvolvido um projeto chamado de **vídeo game**, que simulará um website criado com o objetivo de listar as empresas que fabricam/fabricaram consoles de vídeo game.

Cada empresa será um aplicativo do projeto, que por sua vez terão cadastrados os consoles lançados. Exemplos de aplicativos:

- Nintendo
- Sony
- Microsoft

Como atualmente existem muitos ambientes de desenvolvimento integrado (IDE), não será especificado um para ser usado. Isso fica a cargo do leitor.

Para este tutorial, serão usados os softwares open-source:

- [Visual Studio Code](#)
- [Neovim](#)

Estrutura

A estrutura do projeto será a seguinte:

- **vídeo-game**: será a pasta raiz de todo o projeto; nela serão criadas as pastas do ambiente virtual, do projeto, dos aplicativos etc.;
- **.venv**: será a pasta onde ficará o ambiente virtual criado;
- **videogame**: será o nome da pasta que usada para a criação do projeto; o nome está diferente da pasta raiz para não haver confusão quando aos nomes de cada uma;
- **meus_templates**: nome da pasta que será usada para guardar os templates HTMLs globais do projeto;
- **meus_statics**: nome da pasta que será usada para guardar os arquivos estáticos CSSs, JSs e imagens globais do projeto;
- **nintendo, sony, microsoft**: exemplos de alguns nomes de aplicativos (e suas respectivas pastas) que serão usadas ao longo desse tutorial; elas ficarão na raiz do projeto;

Nomenclatura das Pastas

Sempre que se for referir a uma pasta na raiz do projeto, ela será referenciada apenas com a **/**.

Por exemplo

- se referindo a uma pasta na raiz do projeto: **/nintendo/**
- se referindo a uma pasta em outra pasta: **/nintendo/templates/**
- se referindo a um arquivo em qualquer pasta: **/nintendo/templates/nintendo/paginas/index.html**

Evitando Colisão de Nomes

Para evitar a colisão de nomes, cada aplicativo terá uma pasta chamada de **templates**, que conterá uma pasta interna com o nome do próprio aplicativo. Dentro dessa pasta interna, existirão outras duas pastas, **paginas** e **parciais**.

A **paginas** servirá para guardar os arquivos HTMLs que serão carregados pelos arquivos **views.py** de seus respectivos aplicativos.

A **parciais** guardará os arquivos HTMLs que serão usados para montar as páginas da pasta **paginas**. Esses arquivos HTML nunca serão carregados diretamente, mas apenas importados por outros HTMLs.

Também terão uma pasta chamada de **static**, que servirá para guardar as pastas CSSs, JSs e imagens (e seus respectivos arquivos) de cada aplicativo.

Para as pastas globais (meus_templates e meus_statics), haverá uma pasta interna chamada **global**, que servirá para ser modelo de vários aplicativos, sem pertencer a algum em específico.

Veja como vai ficar as pastas do aplicativo nintendo (o mesmo aplicar-se-á aos demais aplicativos, só mudando o nome da pasta do aplicativo e da pasta abaixo do templates e static):

- /nintendo/templates/nintendo/paginas/
- /nintendo/templates/nintendo/parciais/
- /nintendo/static/nintendo/css/
- /nintendo/static/nintendo/js/
- /nintendo/static/nintendo/imgs/
- /sony/templates/sony/paginas/

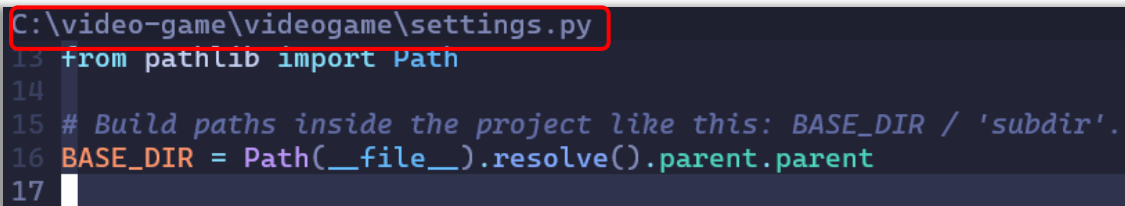
E veja como ficarão as pastas nos modelos globais:

- /meus_templates/global/paginas/
- /meus_templates/global/parciais/
- /meus_statics/global/css/
- /meus_statics/global/js/
- /meus_statics/global/imgs/

Capturas de Tela

Diversas capturas de telas serão usadas para esse tutorial. Sempre que possível, elas serão feitas com o nome do arquivo e sua localização no topo de cada.

Por exemplo:



```
C:\video-game\videogame\settings.py
13 from pathlib import Path
14
15 # Build paths inside the project like this: BASE_DIR / 'subdir'.
16 BASE_DIR = Path(__file__).resolve().parent.parent
17
```

Isso é muito importante, pois diversos arquivos terão o MESMO nome. Logo, atente muito a essa primeira linha para saber onde e qual arquivo está sendo alterado.

Preparando o Local de Trabalho

Ambiente Virtual

Um ambiente virtual é um ambiente isolado em seu sistema operacional que permite que você instale pacotes e gerencie suas dependências de forma separada das outras aplicações em seu computador. Isso é útil porque diferentes aplicações podem exigir diferentes versões de bibliotecas e pacotes, e o uso de um ambiente virtual permite que você mantenha as versões corretas para cada aplicação.

“Um ambiente virtual é um ambiente Python semi-isolado que permite que pacotes sejam instalados para uso por uma aplicação específica, em vez de serem instaladas em todo o sistema.”

[Instalando módulos Python](#)

A partir da versão 3.3 do Python, o [venv](#) é a ferramenta usada por padrão para a criação desses ambientes. Então, ela será a ferramenta usada para a criação e gerência desses ambientes nesse tutorial.

Sobre o venv

“O módulo venv oferece suporte à criação de “ambientes virtuais” leves, cada um com seu próprio conjunto independente de pacotes Python instalados em seus diretórios site. Um ambiente virtual é criado sobre uma instalação existente do Python, conhecido como o Python “base” do ambiente virtual, e pode, opcionalmente, ser isolado dos pacotes no ambiente base, de modo que apenas aqueles explicitamente instalados no ambiente virtual estejam disponíveis.

Quando usadas em um ambiente virtual, ferramentas de instalação comuns, como pip, instalarão pacotes Python em um ambiente virtual sem precisar ser instruído a fazê-lo explicitamente.”

[venv](#)

Criando Ambientes Virtuais

Para criar um ambiente virtual, temos que navegar na pasta que queremos que ele seja criado usando um terminal e executar o comando abaixo:

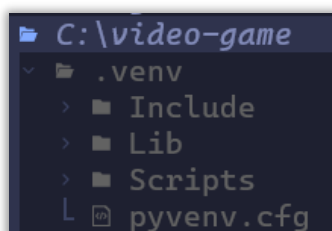
```
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

C:\video-game>python -m venv .venv
```

A execução desse comando cria o diretório de destino e coloca um arquivo pyvenv.cfg nele com uma chave home apontando para a instalação do Python a partir da qual o comando foi executado (um nome comum para o diretório de destino é .venv).

Ele também cria um subdiretório bin (ou Scripts no Windows) que contém uma cópia/link simbólico de binário/binários do Python (conforme apropriado para a plataforma ou argumentos usados no momento da criação do ambiente). Ele também cria um subdiretório (inicialmente vazio) lib/pythonX.Y/site-packages (no Windows, é Lib\site-packages). Se um diretório existente for especificado, ele será reutilizado.

Se olharmos a pasta, veremos que ela foi criada conforme especificado, dentro da nossa pasta vídeo-game:



Essa é a pasta onde ficará nosso ambiente virtual.

Mais conteúdo e explicações sobre as linhas de comando do Python podem ser encontrados em [Linha de Comando e Ambiente](#).

Ativando o Ambiente Virtual

A ativação do ambiente virtual vai depender de seu Sistema Operacional.

A partir da raiz do projeto:

- Windows:
 - `.venv\Scripts\activate`
- Linux / MacOS:
 - `source .venv/bin/activate`

Independente do seu SO, quando ativado pelo terminal, será exibido o nome da pasta do ambiente virtual no começo da linha:

```
C:\video-game>.venv\Scripts\activate  
(.venv) C:\video-game>
```

Isso indica que nosso ambiente virtual está ativo e operante.

Instalando os Pacotes

Agora que o ambiente virtual está ativo e operante, já podemos instalar os pacotes que serão usados.

Para esse tutorial, serão instalados os pacotes do [Django](#), [Faker](#) e [Pillow](#):

```
(.venv) C:\video-game>pip install django faker pillow
```

Após a instalação dos pacotes, pode aparecer uma mensagem notificando que há uma nova versão do pip. Para instalar, basta seguir o comando que aparece na própria notificação do terminal:

```
[notice] A new release of pip available: 22.3.1 → 23.1.2  
[notice] To update, run: python.exe -m pip install --upgrade pip  
(.venv) C:\video-game>python -m pip install --upgrade pip
```

Desativando o Ambiente Virtual

Para desativar o ambiente virtual, basta chamarmos a qualquer momento no terminal o comando deactivate:

```
(.venv) C:\video-game>deactivate  
C:\video-game>
```

Agora, todas as chamadas do Python serão feitas na versão que está instalada juntamente com o SO, e não mais a partir do ambiente virtual.

Criando um Projeto Django

Uma vez instalado o Django, podemos conferir sua versão com o comando abaixo:

```
(.venv) C:\video-game>django-admin --version
4.2.3
```

PS.: Pode acontecer de sua versão ser diferente da que foi mostrada acima. Não se preocupe com isso, já que dificilmente atualizações futuras invalidem versões mais antigas.

Também podemos usar a linha de comando para chamar o help da aplicação e exibir as opções e funcionamentos de determinado comando. Vamos ver um exemplo com o comando runserver:

```
(.venv) C:\video-game>django-admin help runserver
usage: django-admin runserver [-h] [--ipv6] [--nothreading] [--noreload] [--version]
                             [--settings SETTINGS] [--pythonpath PYTHONPATH]
                             [--no-color] [--force-color] [--skip-checks]
                             [addrport]

Starts a lightweight web server for development.

positional arguments:
  addrport              Optional port number, or ipaddr:port

options:
  -h, --help            show this help message and exit
  --ipv6, -6            Tells Django to use an IPv6 address.
  --nothreading         Tells Django to NOT use threading.
  --noreload           Tells Django to NOT use the auto-reloader.
  --version            Show program's version number and exit.
  --settings SETTINGS  The Python path to a settings module, e.g.
                        "myproject.settings.main". If this isn't provided, the
                        DJANGO_SETTINGS_MODULE environment variable will be used.
  --pythonpath PYTHONPATH
                        A directory to add to the Python path, e.g.
                        "/home/djangoprojects/myproject".
  --no-color           Don't colorize the command output.
  --force-color        Force colorization of the command output.
  --skip-checks        Skip system checks.

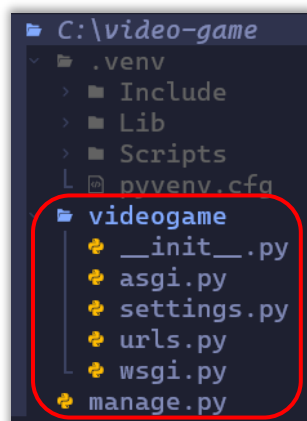
(.venv) C:\video-game>
```

PS.: ao longo do tutorial, mais comandos serão apresentados, então experimente ver a documentação com o help para cada um deles.

Agora, vamos criar um projeto. Para isso, usamos o comando abaixo:

```
(.venv) C:\video-game>django-admin startproject videogame .
```

O comando acima vai criar uma pasta e um arquivo na raiz do nosso projeto. Veja abaixo:



Repare que o comando tem como último parâmetro um ponto. Isso porque ele é usado para definir onde queremos que essa pasta seja criada, logo o ponto indica que queremos criar ele na pasta atual.

Arquivo `manage.py`

O arquivo `manage.py` criado é muito importante durante o desenvolvimento. Tecnicamente, ele faz a mesma coisa que o comando `django-admin`.

Por enquanto, a única finalidade do `django-admin` é a execução do comando `startproject`. Para todos os demais, usaremos o `manage.py`.

Sempre que executarmos o python através dele, é necessário que nosso servidor esteja suspenso. Por exemplo, ao criar um aplicativo.

Pasta do Projeto Criada

A pasta criada (videogame) com o comando anterior, será a pasta com os arquivos de configurações de TODO o nosso projeto. Lá é onde iremos cadastramos novos aplicativos (`settings.py`), definir a localização das novas pastas de nossos templates, arquivos estáticos (CSSs, JSs e imagens em comum a mais de um aplicativo) e mídias (`settings.py`) e criar os links de redirecionamento para cada aplicativo (`urls.py`).

Vamos passar por cada um dos arquivos abaixo:

O arquivo `__init__.py` é o módulo responsável por indicar que aquela pasta é um pacote do python.

Os arquivos `asgi.py` e `wsgi.py` são arquivos de configuração usados quando publicamos nosso site, para realizar a conexão entre o servidor web e o Django. Às vezes um será utilizado, às vezes outro. Vai depender do servidor que nosso site ficará hospedado.

O arquivo `settings.py` é o arquivo de configuração do nosso site Django. Todas as configurações que precisamos para o Django funcionar corretamente precisam estar dentro desse arquivo. Ele é responsável por especificar como o nosso site deve se comportar.

O arquivo `urls.py` é a porta de entrada da nossa aplicação. É onde vamos cadastrar os acessos aos nossos aplicativos (páginas) do site.

Executando e Interrompendo o Servidor

Para iniciar o servidor, basta executar o comando:

```
(.venv) C:\video-game>python manage.py runserver
watching for file changes with StatReloader
Performing system checks ...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the
migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
July 10, 2023 - 13:46:54
Django version 4.2.3, using settings 'videogame.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Uma vez executando, ele vai continuar a ser executado até que seja interrompido manualmente. Para isso, basta pressionar as teclas **Ctrl + c**.

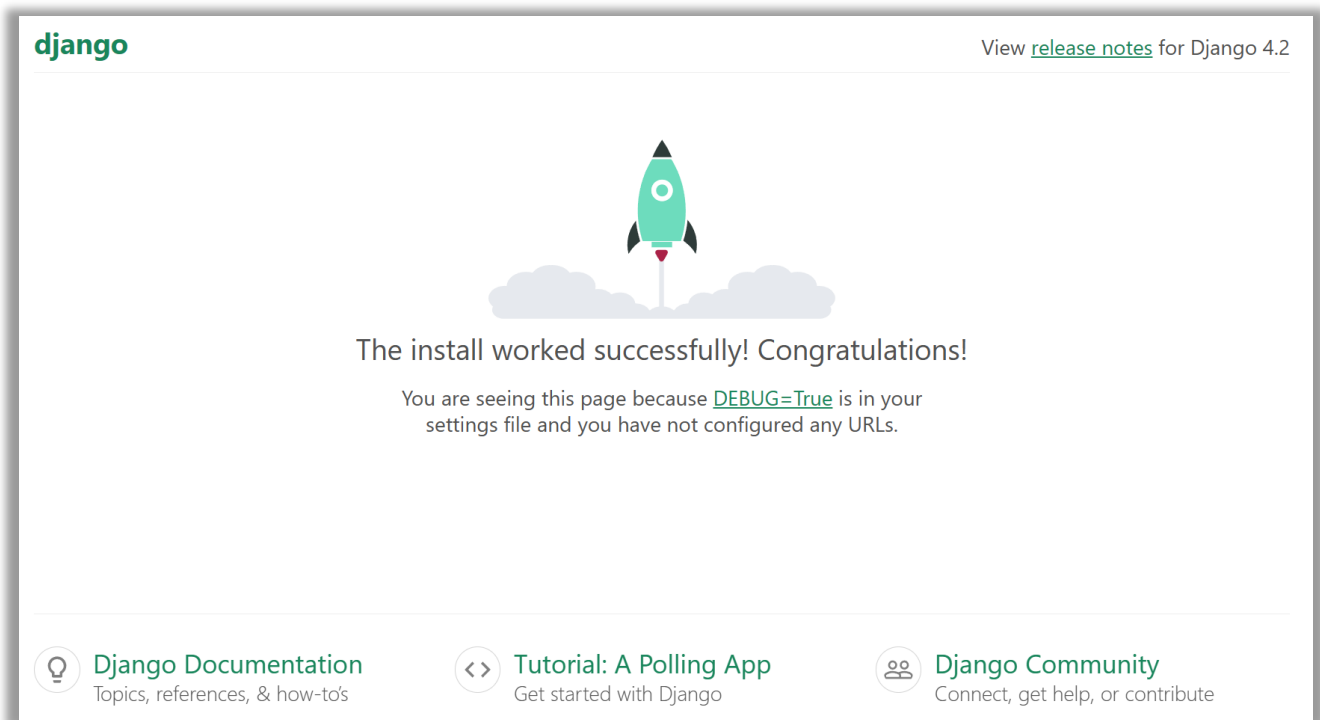
Ao executarmos o projeto pela primeira vez e antes de criar qualquer aplicativo, podemos ver um modelo de uma página de boas-vindas do Django e, ao mesmo tempo, verificar se ele está sendo executado corretamente.

Agora, o nosso site pode ser acessado por duas urls, que na prática levam ao mesmo lugar:

- <http://127.0.0.1:8000/>

- <http://localhost:8000/>

Veja abaixo a página de boas-vindas:



et voilà, temos nossa primeira página!

Arquivo db.sqlite3

Após a execução do servidor pela primeira vez, será criado um arquivo chamado **db.sqlite3**. Ele é o banco de dados que será usado pela aplicação. O nome dele pode ser alterado no arquivo `/videogame/settings.py`:

```
C:\video-game\videogame\settings.py
73 # Database
74 # https://docs.djangoproject.com/en/4.2/ref/settings/#databases
75
76 DATABASES = {
77     'default': {
78         'ENGINE': 'django.db.backends.sqlite3',
79         'NAME': BASE_DIR / 'db.sqlite3',
80     }
81 }
82
```

Esse é o nome padrão do arquivo. Ele não será alterado para esse tutorial.

Configuração Inicial

Agora que nosso projeto já está executando, está na hora de realizarmos as primeiras configurações.

Criando as Tabelas do Banco de Dados

Ao executar o comando runserver, nota-se uma mensagem interessante nela:

```
(.venv) C:\video-game>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks ...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
July 10, 2023 - 14:34:52
Django version 4.2.3, using settings 'videogame.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

O aviso em destaque é porque temos algumas migrações que ainda não foram aplicadas.

Esse aviso indica que o comando para criar as tabelas e registros no banco de dados ainda não foi executado.

Para resolver isso, basta executar o comando mencionado acima e mostrado abaixo:

```
(.venv) C:\video-game>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial ... OK
  Applying auth.0001_initial ... OK
  Applying admin.0001_initial ... OK
  Applying admin.0002_logentry_remove_auto_add ... OK
  Applying admin.0003_logentry_add_action_flag_choices ... OK
  Applying contenttypes.0002_remove_content_type_name ... OK
  Applying auth.0002_alter_permission_name_max_length ... OK
  Applying auth.0003_alter_user_email_max_length ... OK
  Applying auth.0004_alter_user_username_opts ... OK
  Applying auth.0005_alter_user_last_login_null ... OK
  Applying auth.0006_require_contenttypes_0002 ... OK
  Applying auth.0007_alter_validators_add_error_messages ... OK
  Applying auth.0008_alter_user_username_max_length ... OK
  Applying auth.0009_alter_user_last_name_max_length ... OK
  Applying auth.0010_alter_group_name_max_length ... OK
  Applying auth.0011_update_proxy_permissions ... OK
  Applying auth.0012_alter_user_first_name_max_length ... OK
  Applying sessions.0001_initial ... OK

(.venv) C:\video-game>
```

Esse comando vai criar as tabelas de administração, autenticação e sessão do nosso projeto. Agora, é possível criarmos um usuário administrativo.

Usuário Administrativo

Ainda não criamos qualquer aplicativo, mas há um que foi criado por padrão. Sempre que um novo projeto é criado, também é criado o aplicativo **admin**, que conta com um painel para cadastrar, alterar, visualizar e excluir os dados de cada aplicativo. Veremos mais adiante.

Por hora, vamos apenas criar um usuário administrador. Isso só será possível APÓS realizada a primeira migração.

Criando o Usuário admin

Para criar, executamos o comando abaixo:

```
(.venv) C:\video-game>python manage.py createsuperuser
Username (leave blank to use 'gutoh'): admin
Email address: admin@admin.com
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.

(.venv) C:\video-game>
```

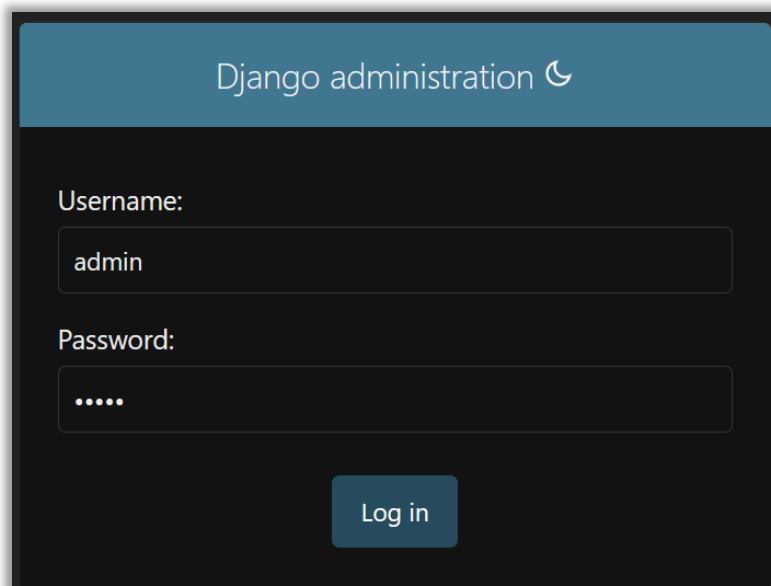
Como usuário, iremos criar um chamado **admin** com a senha também **admin**. Como a senha é muito básica e conhecida, seremos notificados que ela é muito fraca, contudo a usaremos por se tratar de um ambiente de desenvolvimento.

PS.: nunca, jamais, em hipótese alguma use esse usuário e senha em um ambiente de produção. Usaremos aqui por se tratar de um tutorial.

Acessando a Página Administrativa

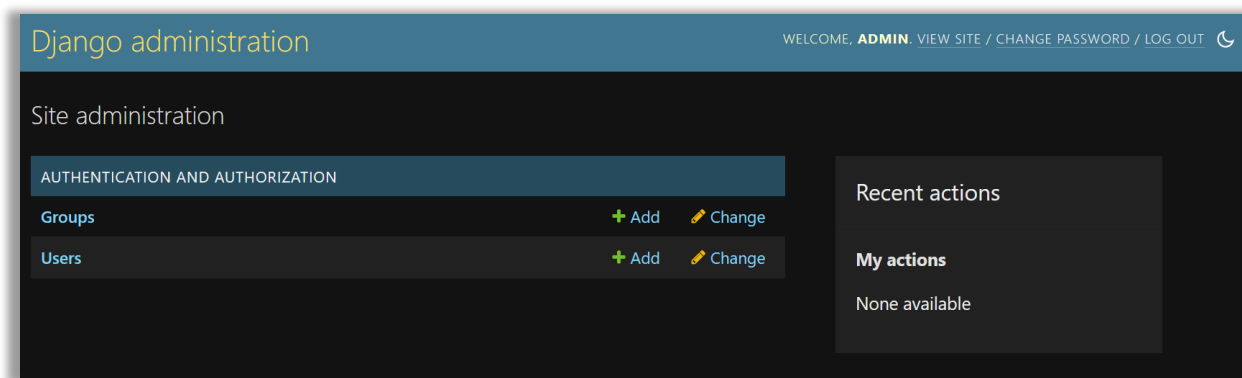
Agora que temos o usuário criado, podemos acessar a página administrativa.

Com o servidor executando, acessamos a página através do link <http://localhost:8000/admin>. Veja a página abaixo:



The image shows the Django administration login interface. It has a dark blue header with the text 'Django administration' and a circular logo. Below the header, there are two input fields: 'Username:' with the value 'admin' and 'Password:' with masked characters '.....'. At the bottom, there is a blue button labeled 'Log in'.

Após acessar, somos apresentados à seguinte página:

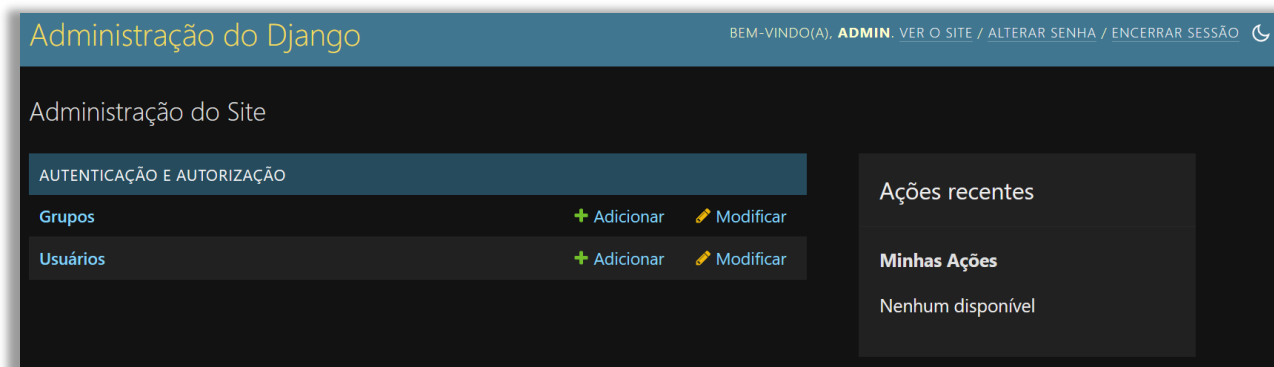


Se observar, toda a interface do site está em inglês. Isso pode ser alterado no arquivo `/videogame/settings.py` através da variável `LANGUAGE_CODE`.

Veja abaixo:

```
C:\video-game\videogame\settings.py
103 # Internationalization
104 # https://docs.djangoproject.com/en/4.2/topics/i18n/
105
106 LANGUAGE_CODE = 'en-us'
107
108 TIME_ZONE = 'UTC'
109
110 USE_I18N = True
111
112 USE_TZ = True
113
```

Se alterarmos para **pt-br**, todo o idioma do nosso site será trocado para o português do Brasil. Veja como ficará a página administrativa agora:



Explore as páginas para se familiarizar com elas.

Como ainda não temos nenhum aplicativo e nenhum modelo criado, não temos mais o que fazer por aqui. Vamos seguir para a criação dos aplicativos.

Criando as Pastas Globais

Ao longo projeto, iremos usar muitos arquivos (principalmente HTML) que serão usados por vários aplicativos.

Por exemplo: cada página inicial de cada aplicativo terá o mesmo layout, logo, é interessante criarmos uma página base para servir de modelo para todas as demais. Assim, se precisarmos atualizar alguma parte da interface, basta alterarmos um arquivo.

Por padrão, essas pastas não vêm configuradas, então cabe ao desenvolvedor realizar essa configuração manualmente. Para isso, algumas partes do arquivo `/videogame/settings.py` precisam ser alteradas.

Primeiro, temos que adicionar uma pasta para guardar esses arquivos HTMLs globais. Ela se chamará `meus_templates`. Para que o Django reconheça essa pasta, temos que adicionar ela na constante `TEMPLATES`, alterando na chave `DIRS` do dicionário de dentro dela (inicialmente será uma lista vazia).

Veja abaixo como ficará:

```
C:\video-game\videogame\settings.py
54 TEMPLATES = [
55     {
56         'BACKEND': 'django.template.backends.django.DjangoTemplates',
57         'DIRS': [
58             BASE_DIR / 'meus_templates',
59         ],
60         'APP_DIRS': True,
61         'OPTIONS': {
62             'context_processors': [
63                 'django.template.context_processors.debug',
64                 'django.template.context_processors.request',
65                 'django.contrib.auth.context_processors.auth',
66                 'django.contrib.messages.context_processors.messages',
67             ],
68         },
69     ],
70 ]
71
```

Depois, temos que especificar as pastas dos arquivos estáticos (CSSs, JSs e imagens) comuns a todos aplicativos. Para isso, adicionaremos mais um pouco de código ao final do arquivo `/videogame/settings.py`.

Veja abaixo:

```
C:\video-game\videogame\settings.py
117 # Static files (CSS, JavaScript, Images)
118 # https://docs.djangoproject.com/en/4.2/howto/static-files/
119
120 STATIC_URL = 'static/'
121
122 STATICFILES_DIRS = [
123     BASE_DIR / 'meus_statics',
124 ]
125 STATIC_ROOT = BASE_DIR / 'static'
126
127 MEDIA_URL = '/midias/'
128 MEDIA_ROOT = BASE_DIR / 'midias'
129
```

Esses nomes das constantes não são aleatórios. O Django, quando executado, busca por essas constantes.

A constante `BASE_DIR` está declarada logo no começo desse mesmo arquivo:

```
C:\video-game\videogame\settings.py
15 # Build paths inside the project like this: BASE_DIR / 'subdir'.
16 BASE_DIR = Path(__file__).resolve().parent.parent
17
```

Ela serve para o Django saber onde que é a raiz do projeto e, então, buscar por todas as pastas a partir dela.

Quando alteramos a chave `DIRS` da constante `TEMPLATES` e adicionamos o nome da pasta, estamos especificando para o Django que há mais uma pasta que servirá para guardar arquivos HTML e deverá ser tratada como as demais existentes nos aplicativos.

A constante `STATICFILES_DIRS` serve para especificar ao Django que há uma pasta com arquivos estáticos na raiz do projeto e deve ser tratada como as pastas `static` de dentro dos aplicativos.

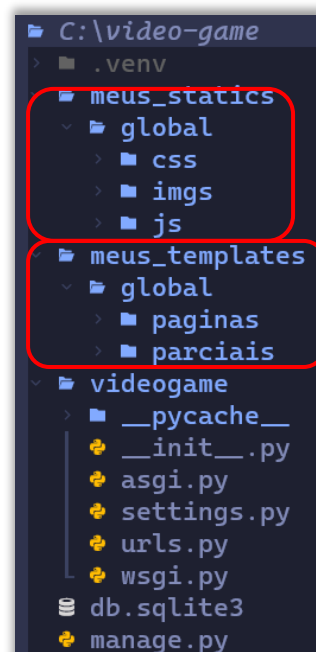
A constante `STATIC_ROOT` serve para especificar onde ficarão os arquivos estáticos após o comando `collectstatic` for executado (veremos mais adiante).

A constante `MEDIA_URL` serve para que o Django saiba onde estão as mídias (fotos e vídeos) do nosso projeto quando tiver que serem carregadas pelas nossas páginas HTMLs.

A constante `MEDIA_ROOT` é a constante que especifica onde essas mídias devem ser salvas ao serem criados os objetos usando a página administrativa.

Agora que especificamos todos os caminhos, temos que criar as pastas que dizemos existir para o Django, com exceção da pasta **mídias** (que será criada automaticamente quando criarmos o primeiro registro na página da administração). Como as pastas `meus_templates` e `meus_statics` serão usadas para guardar os arquivos globais, já vamos criar, também, toda sua estrutura de subpastas.

Veja como vai ficar a nossa organização de pastas do projeto agora:



Repare na indentação das pastas para identificar quem está dentro de quem.

Arquivo urls.py

O arquivo `/videogame/urls.py` é a porta de entrada do nosso site, do nosso projeto.

Quando acessamos qualquer site em Django, estamos acessando primeiro esse arquivo. Ele é o responsável por conectar todos os aplicativos de nosso projeto.

Pense nele como a sala de uma casa. Quando se entra na casa, é o primeiro cômodo. Se quiser ir para um quarto, tem que passar pela sala. Se quiser ir para a cozinha, tem que retornar à sala para então ir ao banheiro. E aí por diante.

Veja como ele é quando criado pelo startproject:

```
C:\video-game\videogame\urls.py
1 """
2 URL configuration for videogame project.
3
4 The 'urlpatterns' list routes URLs to views. For more information please see:
5     https://docs.djangoproject.com/en/4.2/topics/http/urls/
6 Examples:
7 Function views
8     1. Add an import: from my_app import views
9     2. Add a URL to urlpatterns: path('', views.home, name='home')
10 Class-based views
11     1. Add an import: from other_app.views import Home
12     2. Add a URL to urlpatterns: path('', Home.as_view(), name='home')
13 Including another URLconf
14     1. Import the include() function: from django.urls import include, path
15     2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
16 """
17 from django.contrib import admin
18 from django.urls import path
19
20 urlpatterns = [
21     path('admin/', admin.site.urls),
22 ]
```

Observe atentamente os comentários, eles são importantes para ajudar na nossa configuração.

Como ainda não temos aplicativos criados, há apenas uma alteração que precisamos fazer, que é adicionar as referências para as mídias do nosso projeto. Para isso, temos que importar dois módulos e adicionar outro item na variável `urlpatterns`:

```
C:\video-game\videogame\urls.py
17 from django.contrib import admin
18 from django.urls import path
19 from django.conf import settings
20 from django.conf.urls.static import static
21
22 urlpatterns = [
23     path('admin/', admin.site.urls),
24 ]
25
26 urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
27
```

Agora, quando inserirmos as imagens no nosso projeto, o Django já vai saber onde salvar elas e onde buscar elas.

Criando um Aplicativo

Essa etapa será usada diversas vezes sempre que quiser criar um aplicativo para o projeto.

Comando de Criação

Para criar um aplicativo, temos que executar o seguinte comando:

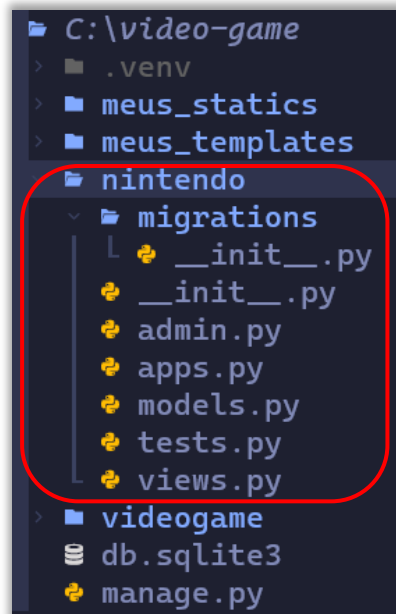
```
(.venv) C:\video-game>python manage.py startapp nintendo
```

IMPORTANTE: todos os aplicativos criados precisam ser uma única palavra, sem o uso de espaços ou qualquer outro caractere especial diferente do sublinhado (como o usado acima).

Por exemplo:

- `python manage.py startapp um_novo_aplicativo;`

Isso vai gerar a seguinte estrutura:



Repare que temos uma nova pasta na raiz do nosso projeto com o nome que especificamos no comando, nintendo. Dentro dela, há diversos módulos python. Temos o `__init__.py`, que faz com que a pasta seja reconhecida como um módulo do python.

A pasta **migrations** será usada para guardar os arquivos gerados no comando `makemigrations`, que veremos em seguida.

Sobre os novos arquivos:

- **admin.py** e **models.py**: servem para o mapeamento objeto-relacional (ORM), que é usado para o banco de dados do aplicativo;
- **apps.py**: nome do seu aplicativo criado, quando alteramos o **settings.py**, vamos usar o valor da variável `name` ou o caminho completo;
- **tests.py**: é usado para testarmos nossa aplicação (não todo o site, mas apenas esse aplicativo);
- **views.py**: arquivo que será usado para criarmos nossas páginas, e será onde colocaremos as funções que chamarão os arquivos HTMLs da pasta **paginas**, também serão usadas para passar variáveis às páginas HTMLs;

Registrando o Aplicativo

Uma vez que o aplicativo está criado, agora temos que registrar ele. Por mais que tenhamos criado o aplicativo e suas pastas, o Django ainda desconhece a existência dele.

Para corrigir isso, temos que adicionar o nome do nosso aplicativo no arquivo `/vídeo-game/settings.py`. Vamos adicionar um novo valor à constante lista chamada **INSTALLED_APPS**.

Para ter certeza desse valor, ele pode ser encontrado no arquivo `/nintendo/apps.py`, conforme pode ser visto abaixo:

```
C:/video-game/nintendo/apps.py
1 from django.apps import AppConfig
2
3
4 class NintendoConfig(AppConfig):
5     default_auto_field = 'django.db.models.BigAutoField'
6     name = 'nintendo'
```

Agora que sabemos exatamente o nome do aplicativo, podemos registrar ele:

```
C:\video-game\videogame\settings.py
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40
41     # meus aplicativos
42     'nintendo',
43 ]
44
```

Agora que o Django já sabe que tal aplicativo existe, já podemos criar as suas páginas.

Como Funcionam as Chamadas

Agora, antes de criarmos as páginas web, isto é, os arquivos HTML, vamos entender como funciona o fluxo de chamada dos arquivos Python e HTML.

Requisição

Quando digitamos <http://localhost:8000/> no campo de url do browser, estamos pedindo ao browser para enviar uma requisição ao nosso servidor.

O servidor, por sua vez, retornará uma página HTML e seu conteúdo estático, isto é, os arquivos CSSs, JSs e imagens. Mas o processo de realizar o retorno desses arquivos é bem mais complexo dentro do servidor executando o Django. O que vai ser descrito abaixo é uma forma um tanto simplificada do seu funcionamento.

Caminho da Requisição

Para essa explicação, será usado o nome do aplicativo da “nintendo”, mas pense nele apenas como modelo. Quando você estiver construindo o seu aplicativo, o nome “nintendo” será diferente.

Por hora, será feita uma explicação superficial. Mais detalhes serão descritos quando os arquivos estiverem sido criados.

Quando o nosso servidor Django recebe a requisição, o primeiro arquivo a ser chamado é o arquivo **/videogame/urls.py**. Através dele, a variável **urlpatterns** vai indicar que o aplicativo **nintendo** tem seu caminho registrados. Em seguida, o arquivo **/nintendo/urls.py** vai ser chamado. Dentro dele haverá novamente a variável **urlpatterns**. Essa variável vai ter a chamada para uma função do arquivo **/nintendo/views.py**. Essa função, é a responsável por realizar a chamada do nosso arquivo HTML, dentro da pasta **/nintendo/templates/nintendo/paginas/index.html**. Caso esse arquivo HTML esteja herdando um layout do **/meus_templates/global/paginas/index_base.html**, ele será chamado posteriormente. Uma vez que o arquivo HTML final for alcançado, será feito esse mesmo caminho na ordem reversa. Quaisquer arquivos CSSs, JSs e imagens que estiverem nos layouts HTML do aplicativo ou no global, serão retornados junto com a página.

Tags Django

Antes de iniciarmos nos arquivos HTML, é importante sabermos o que é e como funcionam as tags Django.

As tags Django são usadas nos arquivos HTMLs carregados pelo Django. Quando o arquivo HTML é lido e uma dessas tags é encontrada, ele é interpretado pelo Django e executada uma determinada ação para uma determinada tag.

Ela tem como característica abrir e fechar com chaves e o símbolo do percentual. Veja abaixo um exemplo da tag include:

A imagem mostra um trecho de código em um editor de texto com fundo escuro. O código é `{% include 'global/parciais/menu.html' %}`. Os caracteres `{%` no início e `%}` no final estão destacados por retângulos vermelhos, indicando a abertura e o fechamento da tag Django.

Neste exemplo, a tag é aberta conforme o destaque na imagem e a tag usada é a include. A maioria das tags tem essa característica, de após a chamada, são usadas aspas para a função.

Sobre o uso de cada tag, conforme elas forem sendo usadas nas imagens, elas serão explicadas.

Criando a Página Web

Agora que sabemos o caminho da chamada dos arquivos (ida e volta), podemos começar a criação dos arquivos HTMLs. Para que não aconteçam erros de criar chamadas para arquivos que não existem, faremos a criação a partir do arquivo **/meus_templates/global/paginas/index_base.html**.

Entendendo os Arquivos Globais

Os arquivos HTML globais serão usados para criar um padrão de layout em todos os nossos aplicativos.

Imagine que temos uma página inicial de cada aplicativo chamada de **/nome_aplicativo/templates/nome_aplicativo/paginas/index.html**. Agora imagine que queremos adicionar uma nota de rodapé em cada um. Se nosso projeto tiver 20 aplicativos, teremos que realizar essa alteração em 20 arquivos diferentes. Isso não é nem um pouco prático. Lembra da filosofia do Django: “don’t repeat yourself”.

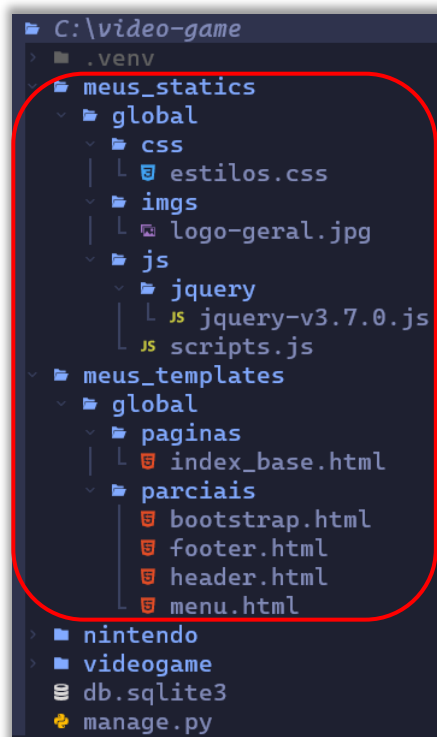
É aí que entram os arquivos globais.

Eles serão construídos para servir de modelo. Agora, vamos construir um arquivo HTML chamado **/meus_templates/global/paginas/index_base.html**. Ele vai conter diversas tags Django chamadas de block. Essas tags block serão usadas para criar blocos que podem (ou não) serem reescritos pelas páginas que herdarão seu layout. Esse processo é chamado de [herança de modelo](#).

Criando os Arquivos Globais

Antes de criar o nosso arquivo HTML global, vamos criar todos os arquivos que serão usados nas nossas pastas **/meus_templates/** e **/meus_statics/**.

Veja como vai ficar:



Repare com MUITA atenção como vai ficar nossa estrutura de arquivos, o que fica em que pasta. Use como referência as linhas verticais para ver o que está dentro do que. Como pode ver, muita coisa foi criada. Muitos arquivos criados.

Para começar a explicar toda essa estrutura, iniciaremos do arquivo **/meus_templates/global/paginas/index_base.html**.

Arquivo HTML Global

Como mencionando anteriormente, agora vamos criar um arquivo HTML que servirá de modelo para TODAS as páginas index dos nossos aplicativos.

Veja como vai ficar o nosso arquivo `/meus_templates/global/paginas/index_base.html`:

```
C:\video-game\meus_templates\global\paginas\index_base.html
1 {% load static %}
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5     <title>{% block app_titulo %}BASE{% endblock app_titulo %}</title>
6     <meta charset="UTF-8">
7     <meta name="viewport" content="width=device-width, initial-scale=1">
8     {% block glb_bootstrap %}
9         {% include 'global/parciais/bootstrap.html' %}
10    {% endblock glb_bootstrap %}
11
12    {% block glb_css %}
13        <link href="{% static 'global/css/estilos.css' %}" rel="stylesheet">
14    {% endblock glb_css %}
15    {% block glb_js %}
16        <script type="text/javascript" src="{% static 'global/js/scripts.js' %}"></script>
17    {% endblock glb_js %}
18
19    {% block app_css_js %}{% endblock app_css_js %}
20 </head>
21 <body>
22     {% block glb_menu %}
23         {% include 'global/parciais/menu.html' %}
24     {% endblock glb_menu %}
25     {% block app_menu %} {% endblock app_menu %}
26
27     {% block app_header %}
28         {% include 'global/parciais/header.html' %}
29     {% endblock app_header %}
30     {% block app_conteudo %}
31         <p>sou um parágrafo gerado pelo /meus_templates/global/paginas/index_base.html</p>
32     {% endblock app_conteudo %}
33     {% block app_footer %}
34         {% include 'global/parciais/footer.html' %}
35     {% endblock app_footer %}
36
37     {% block glb_jquery %}
38         {% include 'global/js/jquery/jquery-v3.7.0.js' %}
39     {% endblock glb_jquery %}
40 </body>
41 </html>
```

Não se assuste com o tamanho dele. Essa é a visão total dele.

Para explicá-lo, vamos passar por trechos menores, um de cada vez.

Mas, uma coisa que já podemos notar logo de cara, mesmo ainda não sabendo exatamente o que cada tag faz (por enquanto), é que há diversas chamadas como:

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

```
{% include 'global/parciais/menu.html' %}
```

Nos exemplos acima, podemos notar claramente a chamada dos arquivos criados nas pastas acima. Repare que todos eles começam com a pasta global.

Isso acontece porque nós especificamos para o Django no `/videogame/settings.py` a existência delas. Então, quando realizamos essas chamadas, o Django já espera encontrar essa estrutura de pastas chamadas.

É por esse motivo que estamos criando uma pasta com o nome do projeto logo dentro da pasta `templates`, `static`, `meus_templates` e `meus_statics`.

Isso traz uma série de benefícios, como:

- o Django sabe exatamente onde encontrar cada arquivo;
- nos permite usar arquivos com o mesmo nome, e ainda manter a distinção entre eles pelos nomes das pastas;

Tag load

Quando trabalhamos com arquivos estáticos (CSS, JS ou imagem) e eles precisam ser carregados para nossas páginas, temos que usar a [tag load](#) no topo do nosso arquivo HTML chamando o `static` (sempre que possível, será a primeira tag na primeira linha do HTML, embora haja uma exceção que veremos adiante). Dessa forma, preparamos o Django para que esses arquivos estáticos sejam carregados corretamente quando usarmos a tag `static`.

```
C:\video-game\meus_templates\global\paginas\index_base.html
1 {% load static %}
2 <!DOCTYPE html>
3 <html lang="en">
```

Tag static

Uma vez que carregamos o comando `static` com a tag `load`, já podemos realizar a chamada desses arquivos nas nossas páginas com a [tag static](#). Veja abaixo:

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

Nessa linha, estamos carregando o arquivo `estilos.css` no caminho especificado. Quando a tag for executada, um caminho será criado para a referência correta do arquivo para o HTML.

Tag include

Quando trabalhamos com a [tag include](#) do Django, o trecho do arquivo chamado vai ser inserido naquele trecho do HTML que o está chamando.

```
{% include 'global/parciais/menu.html' %}
```

Imagine um prédio de 10 andares. Ele tem diversas estruturas que foram incluídas nele, cada uma com seu espaço reservado. Imagine que o elevador dele seja antigo e precise ser substituído por um modelo mais atual. Não é necessário criar um fosso de elevador. O fosso já está lá, o que será trocado (incluído) será apenas o elevador em si.

Tag block

A [tag block](#) tem um comportamento diferente das apresentadas anteriormente. Ela **PRECISA** ter uma tag de fechamento e de um nome, algo que a identifique. Isso porque ela vai ser usada para carregar blocos de código padrão para todas as páginas que a herdarem.


```
{% block app_header %}
    {% include 'global/parciais/header.html' %}
{% endblock app_header %}
{% block app_conteudo %}
    <p>sou um parágrafo gerado pelo /meus_templates/global/paginas/index_base.html</p>
{% endblock app_conteudo %}
{% block app_footer %}
    {% include 'global/parciais/footer.html' %}
{% endblock app_footer %}
```

No exemplo acima (retirada do arquivo **/meus_templates/global/paginas/index_base.html** acima), há 3 blocos. Quando uma página HTML herdar essa página, duas ações podem ser tomadas.

Se algum bloco não for chamado pela página filha, todo o conteúdo dele será carregado por padrão.

Se o bloco for chamado, o seu conteúdo padrão será substituído pelo novo conteúdo da página filha. Logo mais veremos essa segunda parte.

Isso também nos traz outra vantagem, a de ser capaz de não carregar nenhum conteúdo dos blocos, mas veremos mais sobre quando chegarmos na página que a herdará.

Sobre os blocos acima:

- o bloco **app_header** vai carregar o arquivo **/meus_templates/global/parciais/header.html** e todo seu conteúdo para aquele exato ponto do HTML;
- o bloco **app_conteudo**, vai mostrar apenas um parágrafo HTML;
- o bloco **app_footer** vai carregar o arquivo **/meus_templates/global/parciais/footer.html** e todo seu conteúdo;

Você também deve ter reparado que, ao longo da página, há blocos sem conteúdo:

```
{% block app_css_js %}{% endblock app_css_js %}
```

```
{% block app_menu %}{% endblock app_menu %}
```

Elas simplesmente abrem e fecham, sem qualquer conteúdo.

Uma coisa muito importante quando se trabalha com herança de páginas HTML (as tags **block** e **extends**, que veremos adiante) é que as páginas filhas só podem inserir seus conteúdos **DENTRO** das tags bloco herdadas pela página mãe. Então, se quisermos que nossa página seja capaz de carregar conteúdos próprios como, por exemplo, um arquivo CSS e JS, temos que ter blocos designados para tal tarefa.

O trabalho com herança é uma ferramenta muito poderosa, mas também pode se tornar muito complexa se não tomarmos os cuidados para deixar o código legível.