

Framework Django

Por: Augusto Hertzog

Sumário

Framework Django.....	1
Sumário	2
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
Explicando a Explicação	21
Caminho da Requisição.....	21
Criando a Página Web.....	23
Entendendo os Arquivos Globais	23
Criando os Arquivos Globais	23
Arquivo HTML Global	24

Criando as Pastas e Arquivos do Aplicativo	28
Arquivo HTML do Aplicativo	29
Carregando o Arquivo HTML no views.py.....	30
Criando a Chamada da Função no urls.py.....	30
Registrando o Subdomínio do Aplicativo.....	31
A Página.....	31
Novas Páginas	32
Páginas Dinâmicas.....	32
Tags Django	33
Funcionamento da tag	33
Tag load.....	33
Tag static	33
Tag include	34
Tag url	34
Tag block	35
Tag extends	36

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/`

Esse modelo de organização é extremamente útil para organizarmos as chamadas dos arquivos HTMLs nos templates e dos demais arquivos nos static. Dessa forma, evitamos as colisões de nomes ao especificar as chamadas a partir das pastas designadas para os templates e statics.

Por exemplo:

- o arquivo `/nintendo/templates/nintendo/parciais/menu.html` vai ser chamado apenas como `/nintendo/parciais/menu.html`;
- o arquivo `/nintendo/static/nintendo/css/estilos.css` vai ser chamado como `/nintendo/css/estilos.css`;
- o arquivo `/meus_templates/global/paginas/index_base.html` vai ser chamado como `/global/paginas/index_base.html`;

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**.

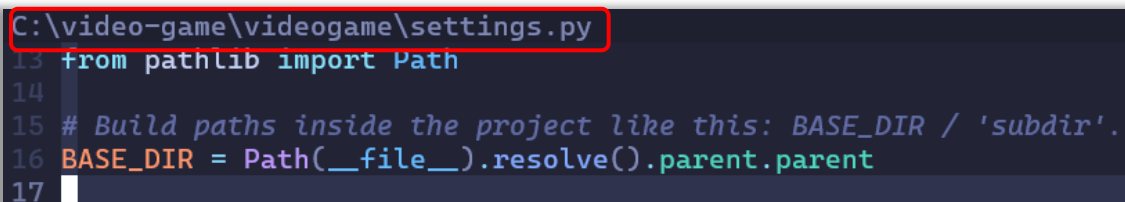
Isso traz uma série de benefícios, como:

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

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:

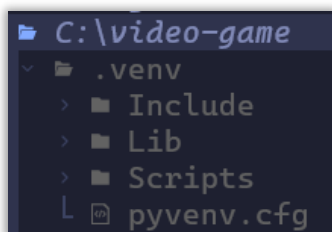
```
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

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.

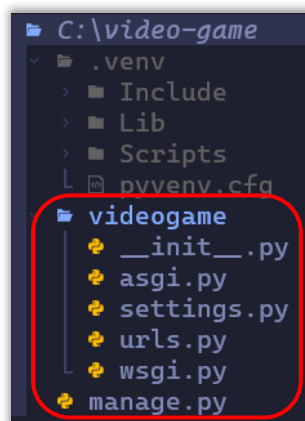
(.venv) C:\video-game>
```

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**.

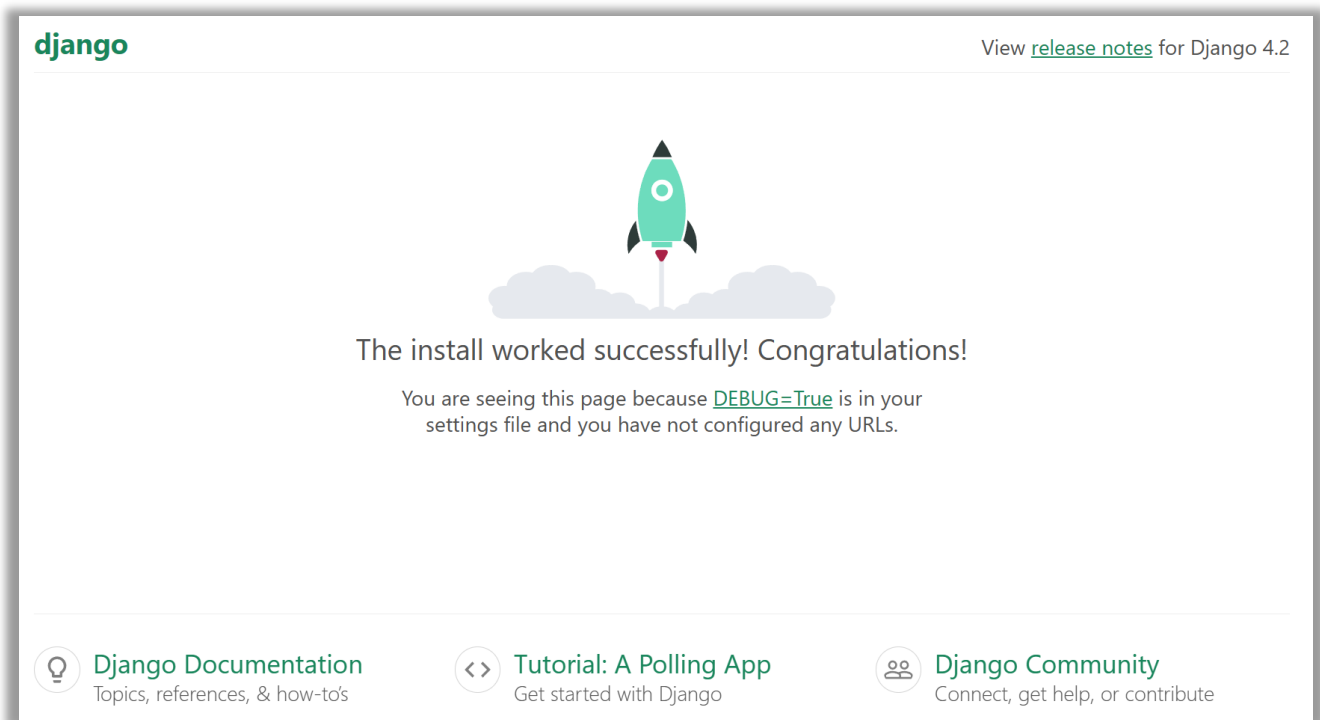
Ao executarmos o projeto pela primeira vez e antes de criar qualquer aplicativo, podemos ver um modelo de uma página de boas-vindas do Django e, ao mesmo tempo, verificar se ele está sendo executado corretamente.

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:

```
(.env) 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:

```
(.env) 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

(.env) 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>
```

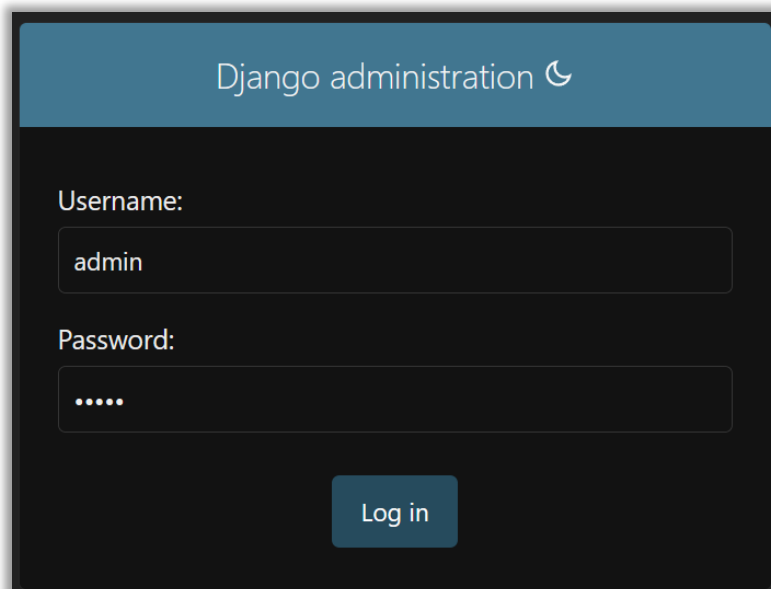
Como usuário, iremos criar um chamado **admin** com a senha também **admin**. Como a senha é muito básica e conhecida, seremos notificados que ela é muito fraca, contudo a usaremos por se tratar de um ambiente de desenvolvimento.

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:



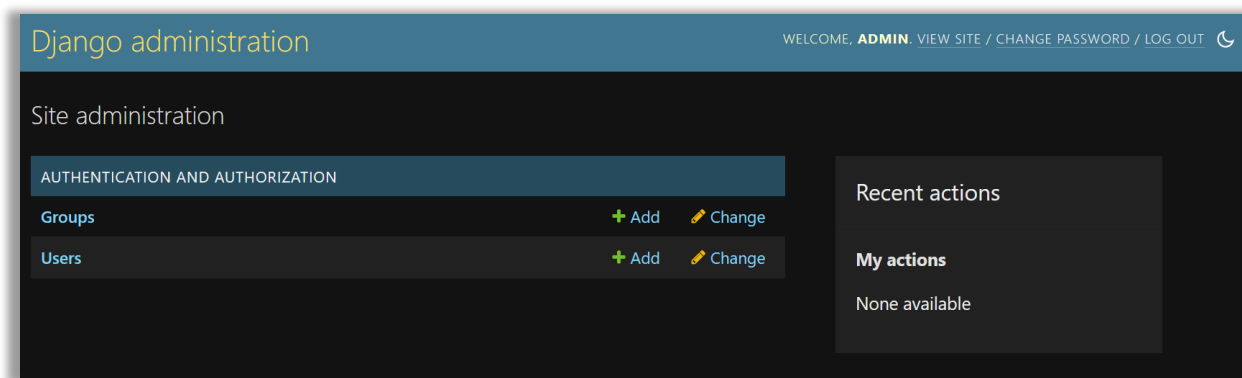
Django administration

Username:
admin

Password:
.....

Log in

Após acessar, somos apresentados à seguinte página:

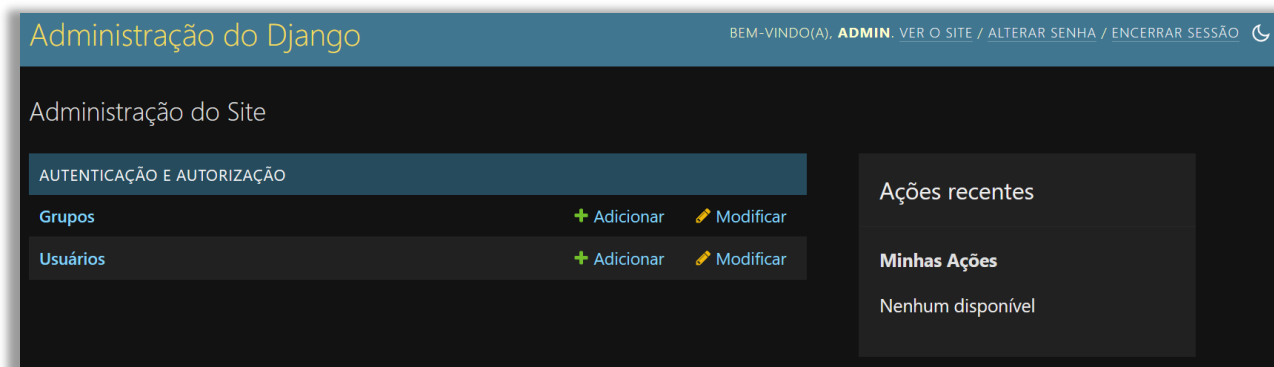


Se observar, toda a interface do site está em inglês. Isso pode ser alterado no arquivo `/videogame/settings.py` através da variável `LANGUAGE_CODE`.

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 (index) 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 esse arquivo localizado em **/meus_templates/global/paginas/**.

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.

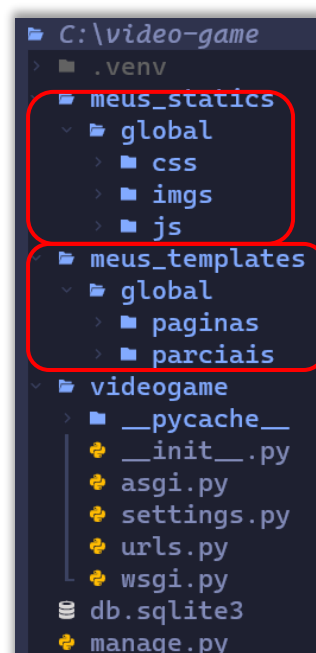
A constante `STATIC_ROOT` serve para especificar onde ficarão os arquivos estáticos após o comando `collectstatic` for executado (veremos mais adiante).

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:

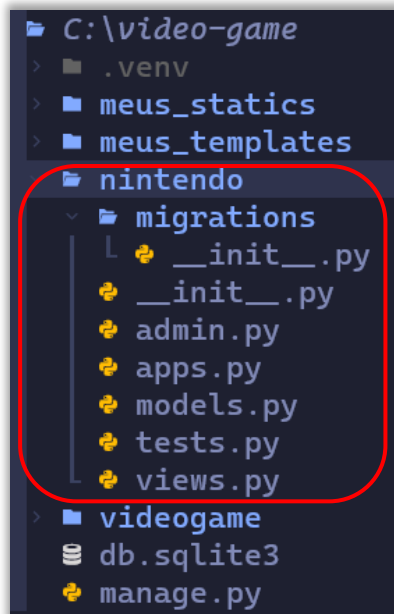
```
(.venv) C:\video-game>python manage.py startapp nintendo
```

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 da requisição do browser para o servidor Django e os arquivos Python e HTML.

Requisição

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

O servidor, por sua vez, retornará uma página HTML e seu conteúdo estático, isto é, os arquivos CSSs, JSs e imagens que porventura estiverem sendo chamados por esse HTML.

Dentro do nosso servidor, acontece uma série de complexos passos para que esses arquivos sejam retornados. Os passos descritos em seguida são uma forma simplificada de todo o processo.

Explicando a Explicação

Para essa explicação, será usado como modelo o aplicativo nintendo e a sua página index como requisição do browser para o servidor, através do link <http://localhost:8000/nintendo/>. Ao final, o mesmo raciocínio pode ser aplicado para os demais aplicativos e suas respectivas páginas.

Por hora, será feita uma explicação superficial, sem especificar as minúcias do funcionamento das funções em cada arquivo. Quando os arquivos forem criados e alterados (`/nintendo/urls.py` e `/nintendo/views.py`, por exemplo), mais detalhes serão fornecidos.

Embora alguns arquivos ainda não existam (como o `/nintendo/urls.py` ou os arquivos HTMLs), eles serão criados quando chegar a hora de falar mais sobre cada um especificamente.

Caminho da Requisição

Passos:

1. o browser envia uma requisição para o servidor Django através da página <http://localhost:8000/nintendo/>;
2. o servidor Django recebe a requisição e a encaminha ao arquivo `/videogame/settings.py`;
3. no arquivo `/videogame/settings.py`:
 - a. a constante `ROOT_URLCONF` indica para qual arquivo deve ser encaminhada a requisição recebida, que nesse caso será o arquivo `/videogame/urls.py`;
4. no arquivo `/videogame/urls.py`:
 - a. a variável `urlpatterns` conterá o registro do subdomínio do aplicativo requisitado (nesse caso, o nintendo) e seu respectivo arquivo `/nintendo/urls.py`;
5. no arquivo `/nintendo/urls.py`:
 - a. a variável `urlpatterns` conterá a chamada para uma função do arquivo `/nintendo/views.py`;
6. no arquivo `/nintendo/views.py`:
 - a. a função é então usada para chamar o arquivo HTML que estará na pasta `/nintendo/templates/nintendo/paginas/index.html`;

Agora, temos duas opções para os arquivos HTML que são chamados pelas funções:

Opção 1:

7. se a página não possuir herança, ela então carrega todas as dependências, que podem ser:
 - a. arquivos HTMLs parciais incluídos com a `tag include` do Django;
 - i. localizados em `/nintendo/templates/nintendo/parciais/`;
 - b. e/ou os arquivos estáticos CSSs, JSs e imagens;
 - i. localizados em `/nintendo/static/nintendo/css/`, `/nintendo/static/nintendo/js` e `/nintendo/static/nintendo/imgs/`;
8. por fim, é feito todo o caminho inverso até o browser;
 - a. passos: $7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$;
9. para então a página ser exibida para o usuário;

Opção 2:

7. se a página estiver herdando outro arquivo HTML (através do uso da tag extends do Django), ela então será chamada da pasta **/meus_templates/global/paginas/**;
8. por fim, esse arquivo **/meus_templates/global/paginas/index_base.html** vai carregar todas as suas dependências, que podem ser:
 - a. arquivos HTMLs parciais incluídos com a tag include do Django;
 - i. localizados em **/meus_templates/global/parciais/**;
 - b. e/ou os arquivos estáticos CSSs, JSs e imagens;
 - i. localizados em **/meus_statics/global/css/**, **/meus_statics/global/js** e **/meus_statics/global/imgs/**;
9. por fim, é feito todo o caminho inverso até o browser;
 - a. passos: 8 → 7 → 6 → 5 → 4 → 3 → 2 → 1;
10. para então a página ser exibida para o usuário;

Sobre o uso específico de cada tag, ao fim do documento, haverá uma seção reservada com explicações e exemplos de cada uma. Sempre que tiver alguma dúvida do funcionamento de uma tag, você poderá descer até o final para consultar seu funcionamento e então retornar para onde parou.

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 ainda não existem, faremos a criação a partir do arquivo mais extremo, o último a ser chamado antes de realizar o caminho de volta, que nesse exemplo será o **/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 (index) 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.

Lembre-se da filosofia do Django:

“don’t repeat yourself”

É aí que entram os arquivos globais.

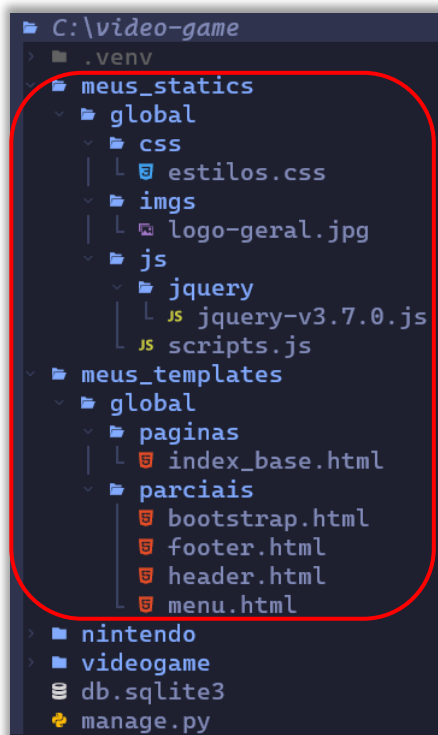
Eles serão construídos para servir de modelo e terão uma estrutura organizacional relativamente genérica (para as páginas que herdarão eles).

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 todas as pastas e 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 tudo isso, 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 um modelo de 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_jquery %}
9         <script type="text/javascript" src="{% static 'global/js/jquery/jquery-v3.7.0.js' %}"></script>
10    {% endblock glb_jquery %}
11    {% block glb_bootstrap %}
12        {% include 'global/parciais/bootstrap.html' %}
13    {% endblock glb_bootstrap %}
14
15    {% block glb_css %}
16        <link href="{% static 'global/css/estilos.css' %}" rel="stylesheet">
17    {% endblock glb_css %}
18    {% block glb_js %}
19        <script type="text/javascript" src="{% static 'global/js/scripts.js' %}"></script>
20    {% endblock glb_js %}
21
22    {% block app_css_js %}{% endblock app_css_js %}
23 </head>
24 <body>
25     {% block glb_menu %}
26         {% include 'global/parciais/menu.html' %}
27     {% endblock glb_menu %}
28     {% block app_menu %}{% endblock app_menu %}
29
30     {% block glb_header %}
31         {% include 'global/parciais/header.html' %}
32     {% endblock glb_header %}
33     {% block app_conteudo %}
34         <p>sou um parágrafo gerado pelo /meus_templates/global/paginas/index_base.html</p>
35     {% endblock app_conteudo %}
36     {% block glb_footer %}
37         {% include 'global/parciais/footer.html' %}
38     {% endblock glb_footer %}
39 </body>
40 </html>
```

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

Para explicá-lo, vamos passar pelas tags e o que cada uma está fazendo, mas uma coisa que já podemos notar logo de cara 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 anteriormente](#). Repare que todos eles começam com a pasta global.

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

Na primeira linha, estamos carregando os modelos estáticos com a [tag load](#), preparando a página para os arquivos estáticos que serão carregados posteriormente:

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

Em seguida, temos a primeira [tag block](#):

```
<title>{% block app_titulo %}BASE{% endblock app_titulo %}</title>
```

Repare que ela está DENTRO da tag HTML title. Isso porque, quando a página que a herdar usar esse bloco para criar o título da página, não será necessário chamar a tag HTML title novamente, basta apenas chamar o bloco `app_titulo` e então definir o que queremos dentro.

Veja um exemplo de uso do bloco `app_titulo` por uma página filha:

```
{% block app_titulo %}Nintendo{% endblock app_titulo %}
```

Nele, o conteúdo do bloco, originalmente BASE, vai ser substituído por Nintendo pela página filha. Também, como as tags HTML title estão ao redor da tag `block app_titulo`, não é necessário chamar elas na página filha para determinar seu título.

Essa é uma boa prática para evitar a repetição de código.

Veja a próxima imagem:

```
{% block glb_jquery %}
<script type="text/javascript" src="{% static 'global/js/jquery/jquery-v3.7.0.js' %}"></script>
{% endblock glb_jquery %}
{% block glb_bootstrap %}
    {% include 'global/parciais/bootstrap.html' %}
{% endblock glb_bootstrap %}
```

Nela, temos as duas bibliotecas usadas pela página global sendo chamadas pela [tag include](#) em blocos distintos. Isso é uma boa prática, já que possibilita as páginas que a herdar terem a opção de não carregar essas bibliotecas.

Bastando para isso, chamar os blocos e os deixar vazios.

O mesmo raciocínio se aplica aos blocos `glb_css` e `glb_js`.

Veja outras imagens:

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

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

Esses dois blocos, declarados na página mãe e vazios, serão usados pelas páginas dos aplicativos que a herdar para realizar as declarações de seus arquivos CSS e JS (para o bloco `app_css_js`) e para o seu menu (para o bloco `app_menu`).

Veja mais uma imagem:

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

Nesse trecho de código, há dois blocos globais usados para incluir na página dois arquivos parciais HTML. Como estão em blocos, há a opção deles não serem carregados pelas suas páginas filhas, enquanto que o bloco `app_conteudo` será usado pela página filha para colocar todo seu conteúdo (para que assim, ela tenha um comportamento similar à tag `body` do HTML).

Agora, veja as imagens de como estão os demais arquivos globais que foram [criados](#) (veja no todo de cada imagem a localização de cada arquivo):

```
C:/video-game/meus_templates/global/parciais/bootstrap.html
1 <!-- Latest compiled and minified CSS -->
2 <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap
  .min.css" integrity="sha384-BVYiISIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u
  " crossorigin="anonymous">
3
4 <!-- Optional theme -->
5 <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap
  -theme.min.css" integrity="sha384-rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQI
  XwL/Sp" crossorigin="anonymous">
6
7 <!-- Latest compiled and minified JavaScript -->
8 <script src="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/js/bootstrap.min.js" integrit
  y="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7L2mCWNIpG9mGCD8wGNlIcPD7Txa" crossorigin="ano
  nymous"></script>
```

Neste arquivo, está apenas sendo carregado as referências do [Bootstrap](#). Ele, então, está sendo incluído no arquivo `/meus_templates/global/paginas/index_base.html`.

```
C:/video-game/meus_templates/global/parciais/menu.html
1 <nav>
2     <ul>
3         <li><a href="{% url 'nintendo:index' %}">Nintendo</a></li>
4         <li><a href="{% url 'admin:index' %}">Admin</a></li>
5     </ul>
6 </nav>
```

Este arquivo parcial está sendo usado para criar um menu geral do site através da [tag url](#) do Django.

```
C:/video-game/meus_templates/global/parciais/header.html
```

```
1 {% load static %}
2 <header>
3     
4     <p>estou sendo gerado pelo /meus_templates/global/parciais/header.html</p>
5 </header>
```

```
C:/video-game/meus_templates/global/parciais/footer.html
```

```
1 <footer>
2     <p>estou sendo gerado pelo /meus_templates/global/parciais/footer.html</p>
3 </footer>
```

```
C:/video-game/meus_statics/global/css/estilos.css
```

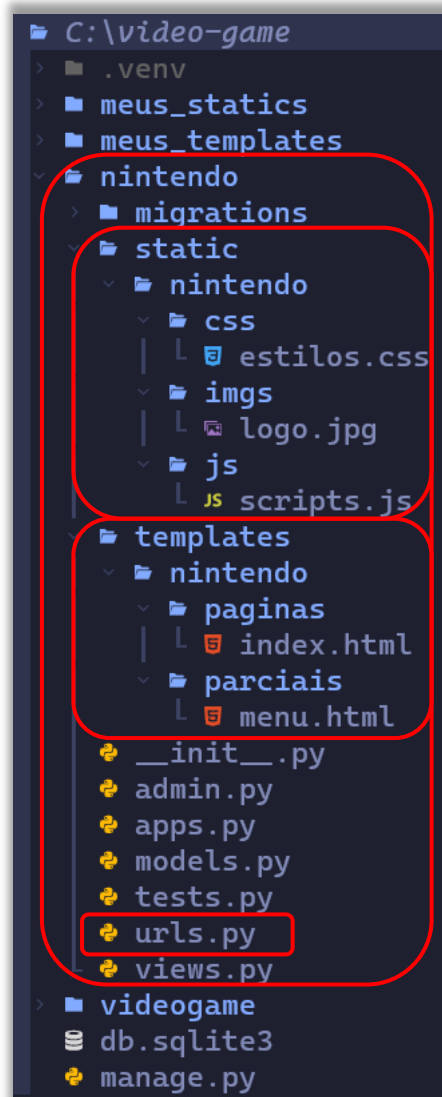
```
1 p{
2     background-color: black;
3     color: white;
4 }
```

O arquivo `/meus_statics/global/js/scripts.js` está vazio, por isso não alguma imagem dele.

Criando as Pastas e Arquivos do Aplicativo

Agora, vamos criar as pastas e arquivos que serão usados pelo nosso aplicativo.

Veja como ficará:



Para o aplicativo, foram criadas as pastas **templates** e **static** (e suas subpastas) e o arquivo **urls.py**, que vai ser o arquivo responsável por gerenciar os subdomínios do aplicativo **nintendo**.

Quando um aplicativo é criado, o Django automaticamente procura pelos arquivos HTMLs dentro da pasta **templates** e os arquivos CSSs, JSs e imagens dentro da pasta **static**. Como esses nomes são padrão para os aplicativos, não há necessidade de registrarmos eles no arquivo **/videogame/settings.py**, assim como fizemos com as pastas **/meus_templates/** e **/meus_statics/**.

A lógica das pastas internas vai ser a mesma da lógica que usamos para as pastas **/meus_templates/** e **/meus_statics/**, isto é, criar a primeira subpasta delas com o mesmo nome do aplicativo para evitar o [Conflito de Nomes](#).

Arquivo HTML do Aplicativo

Agora que temos o arquivo `/nintendo/templates/nintendo/paginas/index.html` criado, vamos usar ele para herdar o layout do `/meus_templates/global/paginas/index_base.html`.

Veja como ele vai ficar:

```
C:\video-game\nintendo\templates\nintendo\paginas\index.html
1 {% extends 'global/paginas/index_base.html' %}
2 {% load static %}
3
4 {% block gbl_bootstrap %}{% endblock gbl_bootstrap %}
5 {% block gbl_js %}{% endblock gbl_js %}
6
7 {% block app_css_js %}
8     <link rel="stylesheet" href="{% static 'nintendo/css/estilos.css' %}">
9 {% endblock app_css_js %}
10
11 {% block app_titulo %}Nintendo{% endblock app_titulo %}
12
13 {% block app_menu %}
14     {% include 'nintendo/parciais/menu.html' %}
15 {% endblock app_menu %}
16
17 {% block app_conteudo %}
18     <p>eu sou um conteúdo sendo carregado do arquivo<br>
19     /nintendo/templates/nintendo/paginas/index.html</p>
20 {% endblock app_conteudo %}
```

Repare como ele fica diferente de um arquivo HTML tradicional.

Logo na primeira linha, há o uso da [tag extends](#) do Django. Isso caracteriza que a página terá um comportamento de herança, que ela herdarà o layout de um outro arquivo HTML.

Repare que alguns blocos são:

- carregados alterando seu conteúdo, como o bloco `app_conteudo`;
- carregados para que seu conteúdo não seja carregado na página, como os blocos `gbl_bootstrap` e `gbl_js`;
- carregados para adicionar um conteúdo, já que originalmente estavam vazios, como o bloco `app_css_js` e `app_menu`;

Veja que as alterações que temos que fazer no arquivo `/nintendo/templates/nintendo/paginas/index.html` são poucas, já que o layout padrão foi definido pelo arquivo `/meus_templates/global/paginas/index_base.html`. Alteramos apenas o que for necessário na página filha.

Veja como ficaria o arquivo, caso esse conceito de herança não tivesse sido aplicado:

```
C:\video-game\nintendo\templates\nintendo\paginas\index.html
1 {% load static %}
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5     <title>Nintendo</title>
6     <meta charset="UTF-8">
7     <meta name="viewport" content="width=device-width, initial-scale=1">
8     <script type="text/javascript" src="{% static 'global/js/jquery/jquery-v3.7.0.js' %}"></script>
9     <link href="{% static 'global/css/estilos.css' %}" rel="stylesheet">
10    <link rel="stylesheet" href="{% static 'nintendo/css/estilos.css' %}">
11 </head>
12 <body>
13     {% include 'global/parciais/menu.html' %}
14     {% include 'nintendo/parciais/menu.html' %}
15     {% include 'global/parciais/header.html' %}
16     <p>eu sou um conteúdo sendo carregado do arquivo<br>
17     /nintendo/templates/nintendo/paginas/index.html</p>
18     {% include 'global/parciais/footer.html' %}
19 </body>
20 </html>
```

Se tivéssemos que realizar a alteração de layout, digamos que queremos adicionar um bloco de código HTML entre as chamadas dos arquivos `/meus_templates/global/parciais/menu.html` e `/nintendo/templates/nintendo/parciais/menu.html` em todos os aplicativos, teríamos que alterar diversos arquivos `index.html`. Enquanto que, aplicando o conceito de herança, bastaria alterar o arquivo `/meus_templates/global/paginas/index_base.html`.

Carregando o Arquivo HTML no views.py

Agora que a página do aplicativo está criada, ela precisa ser carregada através do arquivo `/nintendo/views.py` usando uma função.

Uma boa prática, é criar a função com o mesmo nome da página a ser carregada.

Veja abaixo:

```
C:/video-game/nintendo/views.py
1 from django.shortcuts import render
2
3 def view_index(request):
4     return render(request, 'nintendo/paginas/index.html')
5
```

Acima, temos o arquivo `/nintendo/views.py` e uma função `view_index`. Essa função precisa ser criada com um parâmetro, geralmente chamado de `request` (pois ela será preenchida com a requisição do browser que será recebida).

Essa função precisa, obrigatoriamente, retornar uma chamada da função `render`, que por sua vez (por enquanto), receberá dois parâmetros:

- o primeiro deverá ser a variável `request` recebida pela função;
- a segunda deverá ser o caminho do arquivo HTML a ser carregado, referenciado a partir da pasta `templates` do aplicativo;

Esse arquivo é o que será usado para criar todas as funções que irão carregar todos os arquivos HTMLs presentes na pasta `/nintendo/templates/nintendo/paginas/`. Para futuras páginas, basta adicionar novas funções para as outras páginas.

Criando a Chamada da Função no urls.py

Após a criação da função, temos que chamar ela no arquivo `/nintendo/urls.py`.

Esse arquivo, originalmente, não é criado com a execução do comando `startapp`. Então, é preciso cria-lo manualmente.

A estrutura dele é similar à estrutura existente no arquivo `/videogame/urls.py`, mas com as chamadas para as funções presentes no arquivo `/nintendo/views.py`.

Veja abaixo:

```
C:\video-game\nintendo\urls.py
1 from django.urls import path
2 from . import views
3
4 app_name = 'nintendo'
5
6 urlpatterns = [
7     path('', views.view_index, name='index'),
8 ]
```

Para carregar as funções, é preciso realizar a importação do arquivo **/nintendo/views.py**.

Depois, na chamada da função path, deverá ser passado 3 parâmetros:

- o primeiro é o caminho na url da página; como se trata da página index do aplicativo, deixando em branco, vai ficar com a referência <http://localhost:8000/nintendo/>;
- o segundo é a chamada da função que foi criada no arquivo **/nintendo/views.py**;
- o terceiro é uma string com o nome da página; esse nome será usado pela [tag url](#) para a criação dos links entre as páginas;

Para o registro de futuras páginas, não é necessária a criação do arquivo, pois ele já existe. Para isso, basta adicionar um novo item na lista **urlpatterns** da mesma maneira que foi feito com a página index.html.

Registrando o Subdomínio do Aplicativo

Agora, finalmente, vamos registrar o aplicativo no arquivo **/videogame/urls.py**.

Veja como vai ficar o arquivo:

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

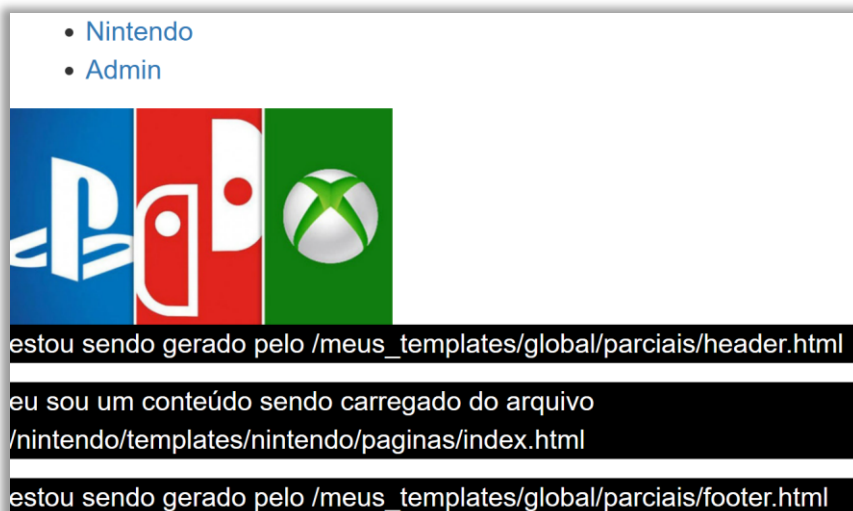
Para registrar um aplicativo no **/videogame/urls.py**, é preciso primeiro importar a função **include**. Depois, criamos um item na lista **urlpatterns** conforme o destaque acima.

Dessa vez, a função path vai ser preenchida com:

- primeiro, será o subdomínio do site que vai pertencer ao aplicativo;
- segundo, será a chamada da função include passando como parâmetro uma string com o nome do aplicativo e o arquivo urls (sem a extensão .py) separados por um ponto;

A Página

Se todos os passos anteriores foram seguidos corretamente, já é possível visualizar a página através do link <http://localhost:8000/nintendo/>:



Novas Páginas

Para o registro de novas páginas, basta seguir os passos desse [capítulo do começo](#).

Páginas Dinâmicas

Tags Django

Este capítulo é todo reservado às tags Django.

Ao longo do tutorial, muitas tags serão apresentadas, mas incluir suas especificações juntamente com outras explicações poderia quebrar o ritmo do tutorial. Então, sempre que uma tag nova for apresentada, será criado um link dela para este capítulo para poder ser realizada a consulta de seu funcionamento.

Funcionamento da tag

As tags Django são usadas nos arquivos HTMLs carregados pelo servidor Django. Quando um arquivo HTML é lido e uma dessas tags é encontrada, ela é interpretada pelo Django e executada uma determinada ação. Cada tag terá funcionalidades diferentes.

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

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

Neste exemplo, a tag é aberta conforme o destaque à esquerda da imagem e fechada conforme o destaque à direita. A tag utilizada é a include. A maioria das tags tem essa característica: dentro delas é especificado qual está sendo chamada e então são usadas aspas para especificar o arquivo relacionado.

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 quando usado com a tag extends). Dessa forma, preparamos o Django para que esses arquivos estáticos sejam carregados corretamente quando usarmos a tag static.

Veja um exemplo:

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

Tag static

Para que uma [tag static](#) funcione corretamente, é necessário carregá-la previamente com a tag load. Uma vez carregada, já podemos realizar a chamada desses arquivos nas nossas páginas com a tag static.

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

Na imagem acima, o caminho completo do arquivo é `/meus_statics/global/css/estilos.css` e, como a pasta `meus_statics` está registrada como pasta para arquivos estáticos, a chamada será somente `/global/css/estilos.css`.

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

Na imagem acima, o caminho completo é `/nintendo/static/nintendo/css/estilos.css`, e, como a pasta static dentro do aplicativo já é detectada automaticamente pelo Django como tal, a chamada será `/nintendo/css/estilos.css`.

Depois de especificado o caminho, a tag static é responsável por localizar o arquivo e montar a referência correta para o arquivo estático. Abaixo, há uma imagem que foi extraída do browser usando a inspeção do código fonte, nela podemos ver o resultado da operação da tag static:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Nintendo</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script type="text/javascript" src="/static/global/js/jquery/jquery-v3.7.0.js"></script>
  <!-- Latest compiled and minified CSS -->
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" integrity="sha384-BVYiISIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  <!-- Optional theme -->
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap-theme.min.css" integrity="sha384-rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQIXwl/Sp" crossorigin="anonymous">
  <!-- Latest compiled and minified JavaScript -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/js/bootstrap.min.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNICPD7Txa" crossorigin="anonymous"></script>
  <link href="/static/global/css/estilos.css" rel="stylesheet">
  <script type="text/javascript" src="/static/global/js/scripts.js"></script>
  <link rel="stylesheet" href="/static/nintendo/css/estilos.css">
</head>
<body>
</body>
</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' %}
```

No exemplo acima, um bloco de código contendo apenas o menu do projeto será inserido onde ele for incluído. Se diversos aplicativos incluírem esse bloco de código e precisarmos adicionar, remover ou atualizar qualquer link do menu, basta alterarmos apenas o arquivo importado e, dessa forma, atualizando todos os arquivos que incluem esse arquivo.

Tag url

A [tag url](#) é usada para o Django criar os links entre os aplicativos. Dentro das aspas vão ser inseridas duas strings separadas por dois pontos:

1. o primeiro representa o nome do aplicativo;
2. o segundo representa o nome da página do aplicativo que deve ser carregada;

Lembrando: esses nomes são definidos nos arquivos de `urls.py` de cada aplicativo.

```
C:/video-game/meus_templates/global/parciais/menu.html
1 <nav>
2   <ul>
3     <li><a href="{% url 'nintendo:index' %}">Nintendo</a></li>
4     <li><a href="{% url 'admin:index' %}">Admin</a></li>
5   </ul>
6 </nav>
```

No exemplo acima, a tag url cria o link para o aplicativo **nintendo** e a página **index**; e depois, o mesmo é feito para o aplicativo **admin** e a sua página **index**.

Tag block

A [tag block](#) tem um comportamento diferente das apresentadas anteriormente.

Ela terá como característica principal a NECESSIDADE de possuir uma tag de fechamento e de um nome personalizado, algo que a identifique. Isso porque ela vai ser usada para carregar blocos de código padrão para todas as páginas que a herdarem.

Outra característica importantíssima dela, é a de que absolutamente NADA pode ser declarado fora de blocos pelas páginas que as herdarem. Se alguma página de algum aplicativo herdar uma página, e tentar incluir um código HTML fora do bloco, esse código será ignorado e não será mostrado no browser.

Veja um exemplo:

```
{% block gbl_footer %}{% endblock gbl_footer %}
<p>jamais irei aparecer no browser</p>
```

Na imagem acima está um trecho de uma página HTML (filha) que está herdando outra página HTML (mãe). Nele, o parágrafo abaixo do bloco jamais será mostrado no browser.

Veja mais um exemplo:

```
{% block gbl_header %}
|   {% include 'global/parciais/header.html' %}
{% endblock gbl_header %}

{% block app_conteudo %}
|   <p>sou um parágrafo gerado pelo /meus_templates/global/paginas/index_base.html</p>
{% endblock app_conteudo %}

{% block gbl_footer %}
|   {% include 'global/parciais/footer.html' %}
{% endblock gbl_footer %}
```

No exemplo acima, há 3 blocos distintos criados em um arquivo HTML global. Quando uma página HTML filha herdar essa página, três ações podem ser tomadas quanto aos blocos herdados:

1. se algum bloco não for chamado pela página filha, todo o conteúdo dentro dele será carregado por padrão pela página filha;
2. se o bloco for chamado pela página filha, e o seu conteúdo alterado, o conteúdo do bloco criado na página mãe será substituído pelo conteúdo do bloco da página filha;

3. se o bloco for chamado pela página filha, mas nenhum conteúdo dentro for especificado, o conteúdo original será substituído pelo conteúdo vazio da página filha. Essa é uma forma de não carregar algum conteúdo herdado, como uma imagem, por exemplo;

Repare que alguns blocos têm como prefixo `gbl_*` ou `app_*`. Esses nomes ajudam a especificar o objetivo de cada bloco. É uma boa prática prefixar os nomes dos blocos que carregam os arquivos globais diferente dos blocos designados especificamente para os aplicativos.

Tag extends

A tag `extends` é usada em conjunto com a tag `block`. Quando ela é usada, ela **DEVE** ser a primeira tag da página filha.

A tag `extends` é usada quando se quer que um arquivo HTML herde o layout de outro arquivo HTML. Quando se faz isso, todo o conteúdo da página mãe herdado pela página filha. Também, toda e qualquer alteração na página filha só poderá ser feita dentro das tags `block` criadas pela página mãe.

Caso a página filha venha a usar a tag `static`, é necessário realizar o carregamento dela novamente usando a tag `load`.