

1 Introdução

Neste material, estudaremos sobre Python e sobre o desenvolvimento com o arcabouço **Django**. Veja algumas de suas características. A página oficial de Django pode ser encontrada em

<https://www.djangoproject.com/>

Nota. Django possui uma filosofia conhecida como “**batteries-included**”. Como veremos a seguir, isso quer dizer que o arcabouço já inclui funcionalidades que muitas aplicações costumam precisar.

Veja algumas de suas principais características.

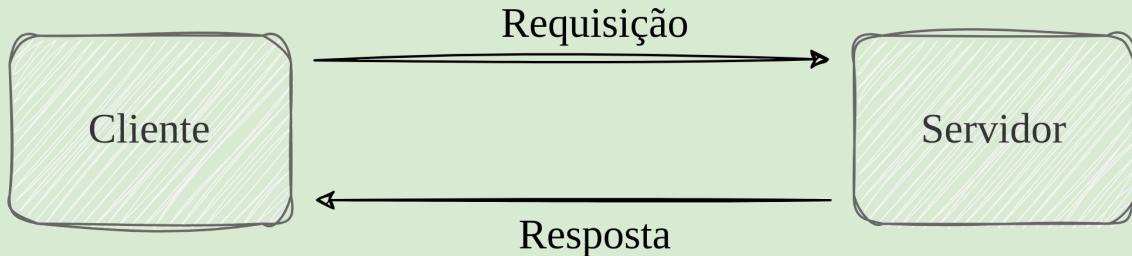
Criação de interface “admin” automática. Em função de metadados incluídos em nosso código, Django cria uma interface gráfica que nos permite manipular conteúdo da aplicação graficamente.

Mapeamento objeto relacional. No que diz respeito ao desenvolvimento de aplicações, o principal paradigma utilizado é aquele baseado na **orientação a objetos**. Por outro lado, o modelo de dados mais utilizado para a persistência de dados ainda é, de longe, o **modelo relacional**. Ou seja, aquele baseado em **tabelas**. Uma aplicação desenvolvida utilizando OO precisa “explicar” como seus dados devem ser armazenados em tabelas. É aí que entra o conhecido **Mapeamento Objeto Relacional (Object Relational Mapping (ORM), em inglês)**. Django inclui construções que trazem alto nível de abstração para resolver este problema.

Arquitetura MTV (Model-Template-View). A arquitetura MTV é muito semelhante ao padrão composto MVC (Model-View-Controller). Em Django,

- **a camada de visualização (produção de HTML, JSON etc)** é implementada por **Templates (T)**. Semelhante à camada View de uma arquitetura MVC.
- **a camada de modelo (representação da base de dados)** é implementada por classes de modelo (**M**).
- **a forma como os dados são apresentados** é decidida na camada **view (V)**. Em geral, essa é uma camada intermediária entre Model e Templates. É semelhante ao Controller de uma arquitetura MVC.
- Em geral, **as regras de negócio podem ser implementadas tanto na view quanto no model**. É um assunto que causa bastante debate. Há uma prática conhecida como “**fat model, thin view**”, que sugere que as regras de negócio ficam principalmente no model, deixando ele mais gordinho.

Uso de funções middleware. Quando desenvolvemos uma aplicação Web, a arquitetura de comunicação predominante é aquela conhecida como **cliente/servidor**. O cliente produz uma requisição e a envia ao servidor. O servidor processa a requisição e produz uma resposta, que é entregue ao cliente.



Django traz construções que viabilizam o uso de conhecidas funções **middleware**. São computações arbitrárias que podemos especificar que ficam “no meio do caminho” entre o cliente e a função alvo no servidor. Antes de a requisição chegar no alvo, ela é processada por uma função middleware.

Nota. Middleware significa algo como “intermediário”.



Uma função middleware pode

- fazer log
- controlar autenticação e autorização
- fazer pré processamento da requisição, como transformá-la para uma forma mais apropriada, esperada pelo servidor.

Manipulação de forms e validação de campos. Django possui classes e funções próprias para a manipulação de forms. Podemos, por exemplo, definir uma classe de modelo e deixá-la “vinculada” a um form, junto com regras de validação. Utilizando Django, a execução das validações e a construção de um objeto de modelo a partir dos dados do formulário são feitas em alto nível de abstração.

Suporte à autenticação e à autorização. Autenticação e autorização são requisitos básicos de praticamente qualquer aplicação. Django inclui um sistema para tal, constituído por views, forms, classes de modelo, controle de login, logout, recuperação de senha etc.

Roteamento de URLs. Mecanismo baseado em expressões regulares para direcionar uma requisição ao componente correto, de acordo com a URL de acesso.

Engine para a geração de templates. Caso deseje construir a camada de visualização produzindo HTML do lado do servidor, Django possui um mecanismo simples para isso.

Melhoria de desempenho com cache. Diferentes formas para fazer “cache” de conteúdo da aplicação o que, em geral, traz benefícios do ponto de vista do desempenho.

Internacionalização. Suporte para a construção de aplicações que utilizam múltiplos idiomas, simplificando a troca de idiomas de acordo com a preferência do usuário.

Signals: Programação baseada em eventos. O sistema “Signals” de Django permite que uma função qualquer seja executada mediante a ocorrência de um evento de interesse. Ele pode ser interessante para

- realização de logs
- interação com aplicações externas
- envio de notificações por e-mail

Servidor Web de teste embutido. Django inclui um servidor web simples, próprio para o teste de aplicações em tempo de desenvolvimento.

Database migrations. Trata-se de um sistema que permite que o “schema” (coleção de tabelas) do banco de dados evolua conforme a aplicação evolui, além da manutenção de um histórico de versões que permite “migrar” de uma a outra, conforme a necessidade.

Documentação. Django foi criado por desenvolvedores do jornal impresso Lawrence Journal-World, que circula em Kansas, nos Estados Unidos. Dizem que, por ter sido criado dentro de um jornal, a sua documentação é muito bem escrita e organizada.

Neste material, vamos desenvolver uma aplicação com Django que ilustra as suas principais funcionalidades.

Nota. Quando trabalhamos com Django, criamos projetos e aplicações.

Um **projeto** é uma coleção de aplicações e de configurações que garantem o seu funcionamento.

Uma aplicação está contida em um projeto. Ela implementa uma funcionalidade de interesse. Por exemplo, em um mesmo projeto Django podemos ter uma aplicação que lida com autenticação, outra que lida com um CRUD de livros, outra que lida com um CRUD de comentários, outra que lida com acessos ao ChatGPT e assim por diante. Essa arquitetura dá origem à expressão “pluggable apps”. A ideia é que uma aplicação é uma pequena criatura que pode ser plugada em e desplugada de diferentes projetos, o que promove **alto nível de reusabilidade**.

2 Desenvolvimento

2.1 (Nova pasta, Ambiente Virtual Python, Novo projeto Django, VS Code) Crie uma pasta para abrigar seus projetos Django. No Windows, uma sugestão é

C:\Users\usuario\Documents\dev\django

Em sistemas Unix-like, use

/home/usuario/dev/django

Abra um terminal (CMD no Windows, Bash em sistemas Unix-like) e navegue até o diretório recém-criado.

```
cd C:\Users\usuario\Documents\dev\django //Windows
cd /home/usuario/dev/django //Unix-like
```

Use

python -m venv venv

para criar um ambiente virtual Python chamado **venv** utilizando o módulo **venv**. Após a sua execução, uma pasta chamada **venv** deve ser criada na raiz de seu projeto.

Nota. Um ambiente virtual Python permite o uso de uma versão específica do Python e também de pacotes diversos. Isso permite que diferentes projetos Python com dependências de pacotes de versões diferentes tenham vida sem impactar uns aos outros, operando de maneira isolada.

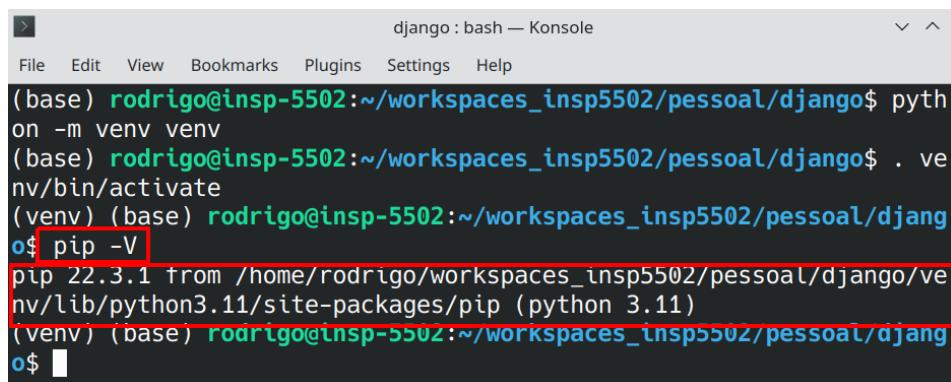
Depois da criação do ambiente virtual, é preciso ativá-lo. Usuários Windows, podem utilizar um terminal “cmd” ou um terminal “Powershell”. A tabela a seguir resume a forma como o ambiente virtual Python pode ser ativado em qualquer caso.

Terminal	Comando(s)
Windows cmd	venv\Scripts\activate.bat
Windows Powershell	Set-ExecutionPolicy -Scope CurrentUser unrestricted venv\Scripts\Activate.ps1
Unix-like terminals	. venv/bin/activate

Em qualquer caso, você pode verificar se o ambiente foi ativado com sucesso com

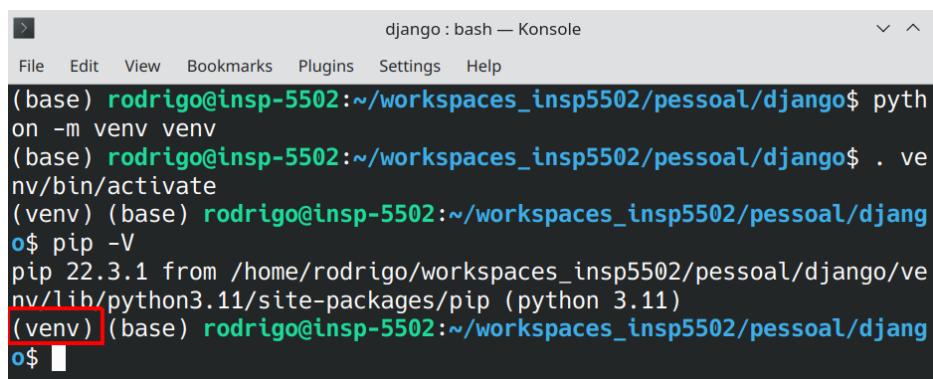
pip -V

A pasta de seu ambiente virtual deve ser exibida.



```
django : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
(base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ python -m venv venv
(base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ . venv/bin/activate
(venv) (base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ pip -V
pip 22.3.1 from /home/rodrigo/worksheets_insp5502/pessoal/django/venv/lib/python3.11/site-packages/pip (python 3.11)
(venv) (base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$
```

O nome do ambiente virtual que você criou também deve aparecer.



```
django : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
(base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ python -m venv venv
(base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ . venv/bin/activate
(venv) (base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$ pip -V
pip 22.3.1 from /home/rodrigo/worksheets_insp5502/pessoal/django/venv/lib/python3.11/site-packages/pip (python 3.11)
(venv) (base) rodrigo@insp-5502:~/worksheets_insp5502/pessoal/django$
```

A seguir, podemos instalar o pacote Django. Essa instalação será válida apenas para o ambiente virtual Python ativo no momento.

```
pip install django
```

Assim, temos um ambiente virtual Python que já inclui o Django, que poderemos utilizar sempre que necessário.

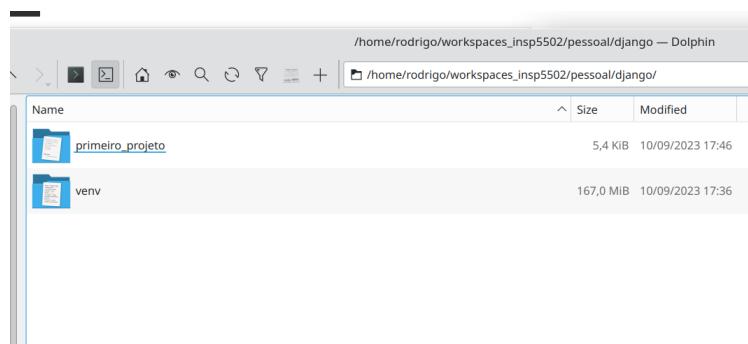
O próximo passo é criar um **projeto Django**. Um projeto Django é uma coleção de aplicações Django e configurações. Em geral, ele possui

- um “CLI” (command line interface) que nos permite interagir com seu conteúdo
- arquivos de configurações
- arquivos de definição de URLs

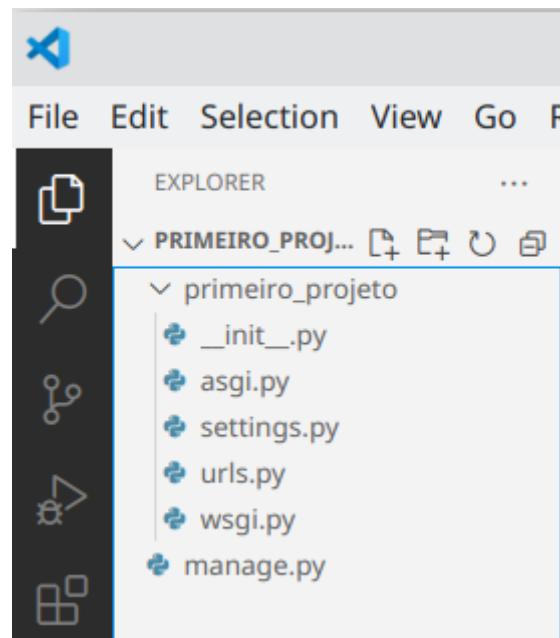
Podemos criar o projeto com

```
django-admin startproject primeiro_projeto
```

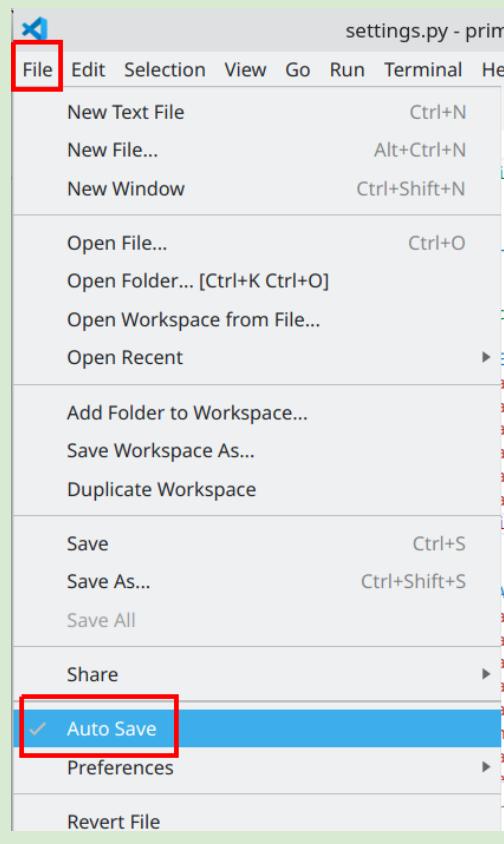
Uma pasta deve ter sido criada, ao lado da pasta que representa o seu ambiente virtual.



Abra o VS Code e clique em **File > Open Folder**. Navegue para encontrar a pasta de seu projeto e vincule o VS Code a ela. Veja o resultado esperado.



Nota. É recomendável manter o VS Code em modo de salvamento automático, clicando em File >> Auto Save.



Veja uma breve explicação sobre cada arquivo.

- **__init__.py**: Um arquivo Python vazio. Seu nome é especial e indica ao ambiente Python que este diretório é um pacote ou módulo Python.

Nota. Em projetos Python, ainda que fora do contexto Django, a existência de um arquivo `__init__.py` dentro de um diretório o caracteriza como um módulo que pode ser importado por outros módulos. Em geral, este diretório tem diversos outros arquivos `.py` que podem ser executados e é bastante comum que o arquivo `__init__.py` permaneça vazio.

- **asgi.py**: ASGI significa Asynchronous Server Gateway Interface. Fornece uma interface padrão entre aplicações Python assíncronas. Veja mais sobre ASGI em

<https://asgi.readthedocs.io/en/latest/>

- **settings.py**: Configurações do projeto, como funções Middleware, aplicações que fazem parte do projeto etc.
- **urls.py**: Neste arquivo, armazenamos os mapeamentos de URL/funções e URL/classes de nossa aplicação. Ou seja, dado um acesso a uma URL, qual funcionalidade de nosso projeto deve ser acionada?
- **wsgi**: WSGI significa Web Server Gateway Interface. É uma especificação que diz como um servidor web “conversa” com aplicações web e como aplicações web podem ser encadeadas para lidar com uma requisição. **Importante para a fase de implantação da aplicação.** Veja mais sobre WSGI em

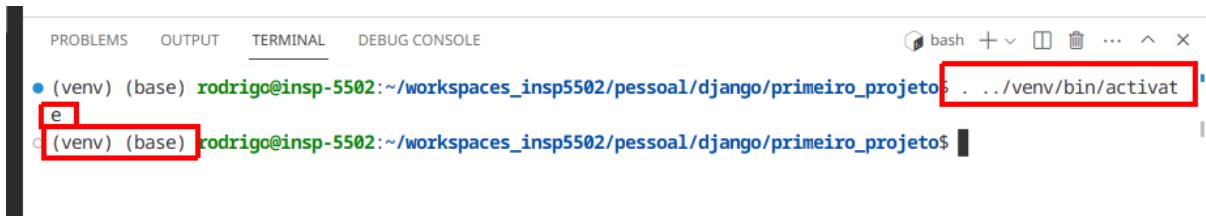
<https://wsgi.readthedocs.io/en/latest/what.html>

- **manage.py**: Contém uma “CLI” (Command line interface) que utilizaremos para realizar tarefas diversas conforme desenvolvemos.

No VS Code, clique em **Terminal >> New terminal** para abrir um terminal interno do VS Code. Lembre-se de **ativar o ambiente virtual que criamos anteriormente**, assim ele será válido para essa instância de terminal que acabamos de abrir.

Cuidado. O ambiente virtual se encontra numa pasta chamada **venv** que está um nível acima da pasta em que estamos no momento. Por isso, use `../` para acessá-la.

Veja o resultado esperado.



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
● (venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/primeiro_projeto$ . . . /venv/bin/activate
e
● (venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/primeiro_projeto$
```

Use o comando apropriado para o seu sistema operacional.

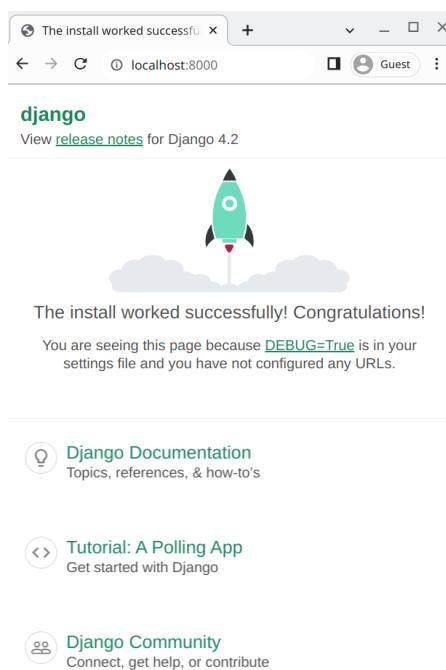
2.2(Executando um servidor local e migrations das aplicações padrão) Use

`python manage.py runserver`

para colocar um servidor em execução. A seguir, visite

`localhost:8000`

usando uma aba de seu navegador para ver algo parecido com



Nota. Se desejar utilizar outra porta, como a 8080, execute

`python manage.py runserver 8080`

Aqui, vamos manter o uso da porta padrão, a 8000.

Observe a mensagem que aparece no terminal.

manage.py - primeiro_projeto - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER PRIMEIRO_PROJETO

PRIMEIRO_PROJETO

primeiro_projeto

> __pycache__

__init__.py

asgi.py

settings.py

urls.py

wsgi.py

db.sqlite3

manage.py

manage.py 1

manage.py ...

```
1 #!/usr/bin/env python
2 """Django's command-line utility for administrative tasks.
3 """
4 import os
5 import sys
6
7 def main():
8     """Run administrative tasks."""
9     os.environ.setdefault('DJANGO_SETTINGS_MODULE',
10                          'primeiro_projeto.settings')
11     try:
12         from django.core.management import
13         execute_from_command_line
14     except ImportError as exc:
15         raise ImportError(
16             "Couldn't import Django. Are you sure it's"
17             "installed and "
18             "available on "
19             "your PYTHONPATH environment variable? Did you "
20             "forget to add "
21             "primeiro_projeto to your PYTHONPATH?"
22         )
23     execute_from_command_line(sys.argv)
```

PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE

(venv) (base) rodrigoeins-5502:~/workspaces_insp5502/pessoal/django/primeiro_px

objeto[s] .../venv/bin/activate

(venv) (base) rodrigoeins-5502:~/workspaces_insp5502/pessoal/django/primeiro_px

(venv) (base) rodrigoeins-5502:~/workspaces_insp5502/pessoal/django/primeiro_px

objeto[s] p

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.

Sept 10, 2023 21:19:59

Django version 4.2.5, using settings 'primeiro_projeto.settings'.

Starting development server at http://127.0.0.1:8000/

Quit the server with CONTROL-C.

[10/Sep/2023 21:18:19] "GET / HTTP/1.1" 200 10664

Not Found: /favicon.ico

[10/Sep/2023 21:18:19] "GET /favicon.ico HTTP/1.1" 404 2120

[10/Sep/2023 21:23:02] "GET / HTTP/1.1" 200 10664

OUTLINE

TIMELINE

Por padrão, o projeto que criamos já inclui diversas aplicações prontas, como **admin**, **auth**, **contenttypes** e **sessions**. Elas estão relacionadas à filosofia “batteries-included” do Django, pois oferecem funcionalidades que, em geral, muitas aplicações precisam. Elas possuem as suas próprias bases de dados e a mensagem diz que há novos modelos de dados para elas, os quais podem ser obtidos por meio da realização de uma migração. No terminal, encerre a execução do servidor com **CTRL+C**. A seguir, use

```
python manage.py migrate
```

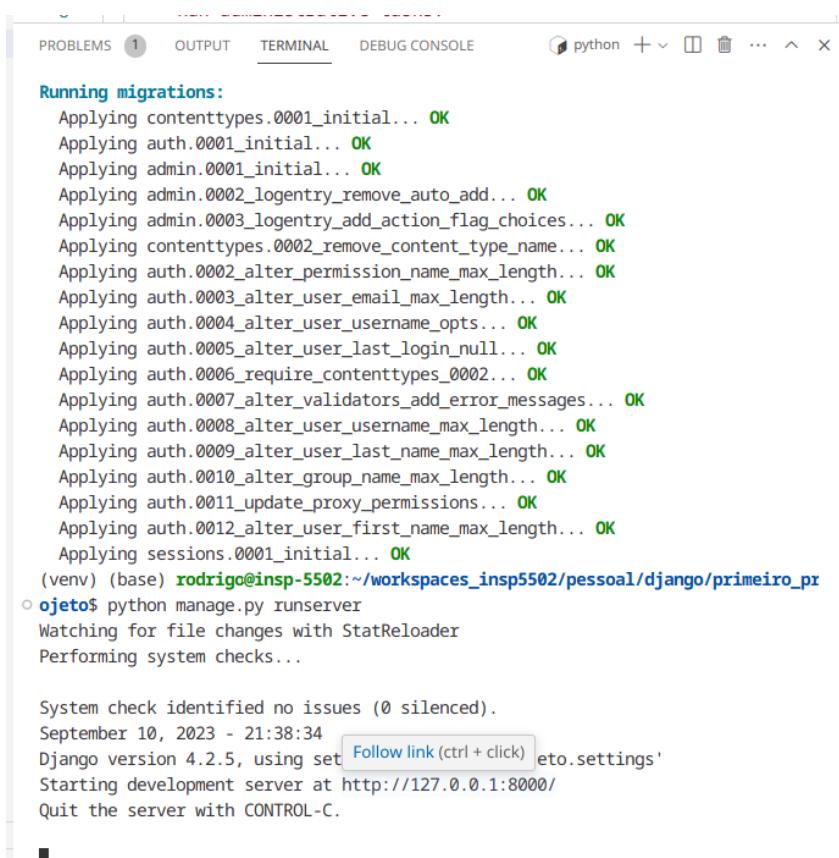
para fazer as migrações.

Execute o servidor novamente com

```
python manage.py runserver
```

Observe que a mensagem sobre migrações não deve mais aparecer.

Nota. Acabamos de criar nosso primeiro projeto Django e sequer conhecemos tais aplicações que já estão incluídas nele. Assim, a migração de base de dados feita no momento pode não fazer muito sentido. Aprenderemos mais sobre isso no futuro.



```
PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE python + - x

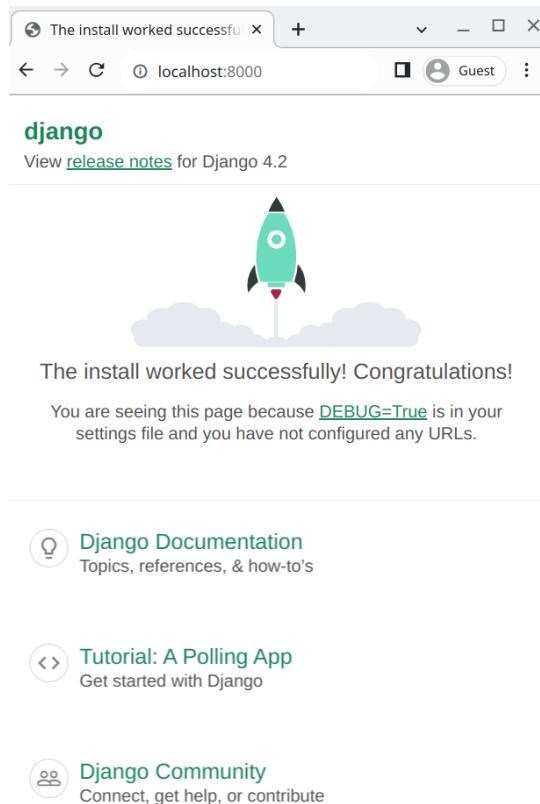
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... 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) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/primeiro_pr
o objeto$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 10, 2023 - 21:38:34
Django version 4.2.5, using settings 'base.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

No seu navegador, visite novamente

localhost:8000

para certificar-se de que está tudo ok.



2.3 (Nova aplicação Django) Agora vamos criar uma aplicação Django que fará parte do projeto atual. Lembre-se: um projeto Django é uma coleção de aplicações, além de configurações diversas que garantem o seu funcionamento. Para criar uma aplicação, use

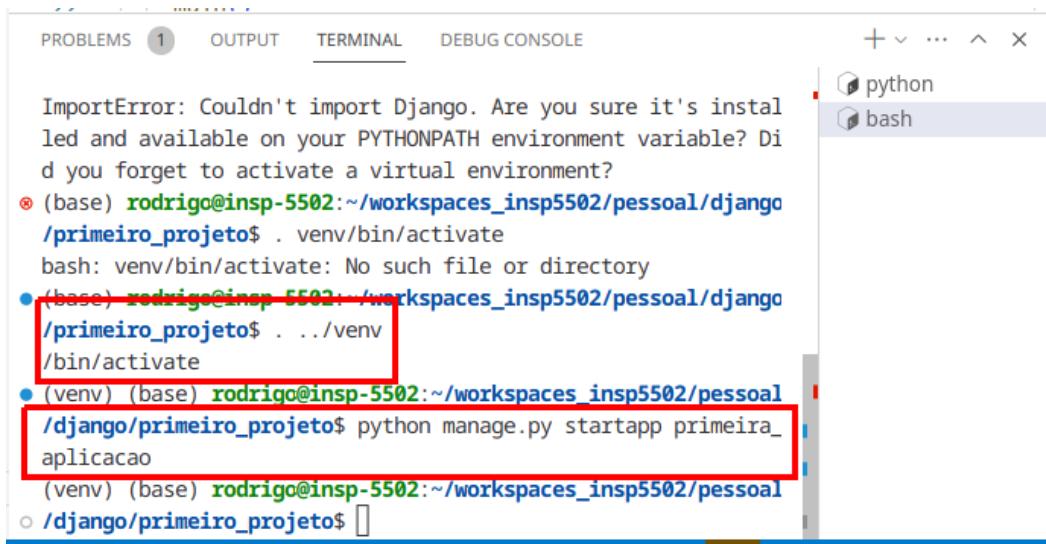
```
python manage.py startapp primeira_aplicacao
```

Para isso, se desejar manter o servidor em execução, abra novo terminal no VS Code.

```
PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE
+ ... ^ x
python
bash

ImportError: Couldn't import Django. Are you sure it's installed and available on your PYTHONPATH environment variable? Did you forget to activate a virtual environment?
(base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django
/primeiro_projeto$ . venv/bin/activate
bash: venv/bin/activate: No such file or directory
(base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django
/primeiro_projeto$ . ./.venv
/bin/activate
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal
/django/primeiro_projeto$ python manage.py startapp primeira_
aplicacao
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal
/django/primeiro_projeto$
```

Se fizer desta forma, com o novo terminal, lembre-se de ativar o ambiente virtual Python para ele também.



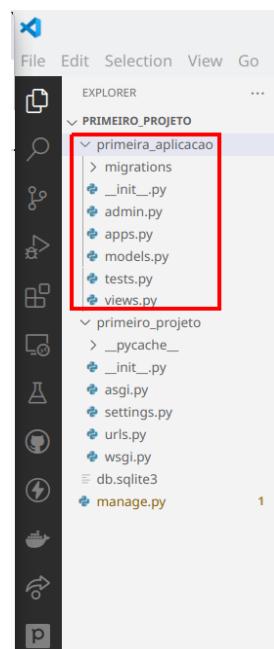
```
PROBLEMS 1 OUTPUT TERMINAL DEBUG CONSOLE

ImportError: Couldn't import Django. Are you sure it's installed and available on your PYTHONPATH environment variable? Did you forget to activate a virtual environment?
① (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django
  /primeiro_projeto$ . venv/bin/activate
  bash: venv/bin/activate: No such file or directory
● (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django
  /primeiro_projeto$ . . ./venv
  /bin/activate
● (venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal
  /django/primeiro_projeto$ python manage.py startapp primeira_
  aplicacao
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal
  ○ /django/primeiro_projeto$
```

The terminal window shows the creation of a Django application named 'primeira_aplicacao'. The 'bash' tab is selected in the top right. The terminal output is as follows:

- ImportError: Couldn't import Django. Are you sure it's installed and available on your PYTHONPATH environment variable? Did you forget to activate a virtual environment?
- (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django /primeiro_projeto\$. venv/bin/activate
- bash: venv/bin/activate: No such file or directory
- (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal /primeiro_projeto\$. . ./venv /bin/activate
- (venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal /django/primeiro_projeto\$ python manage.py startapp primeira_aplicacao
- (venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal ○ /django/primeiro_projeto\$

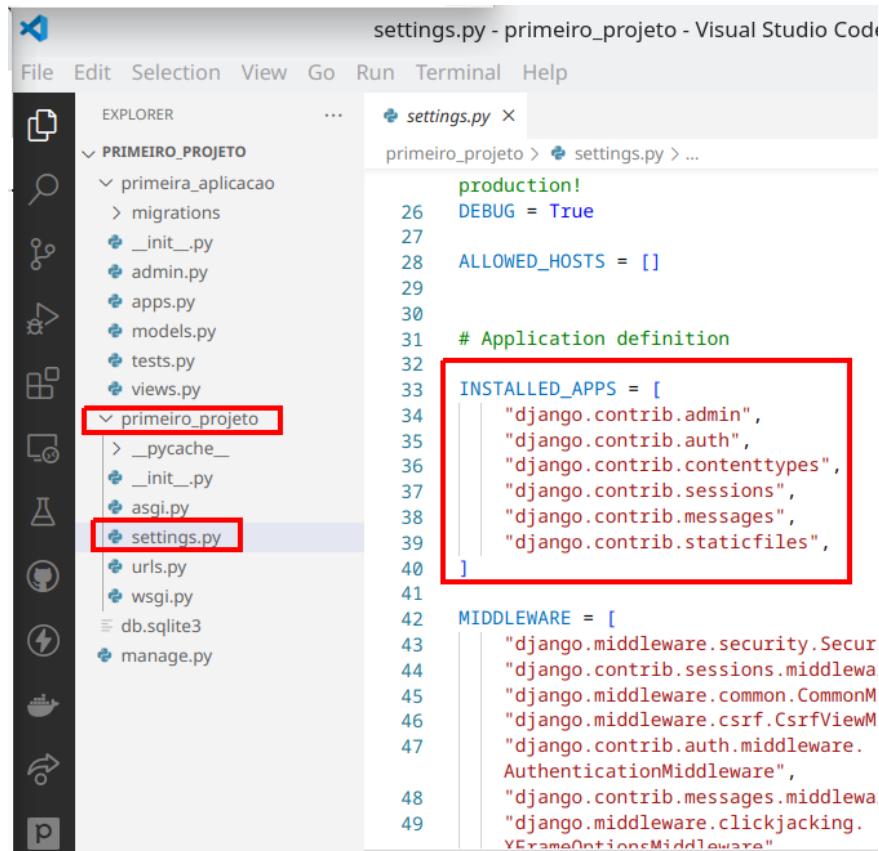
Veja o resultado esperado.



Estudemos o significado de cada arquivo/pasta de uma aplicação Django.

- **__init__.py**: um arquivo vazio que indica que este diretório é um pacote/módulo Python. Quando executarmos este módulo, este arquivo entrará em execução.
- **admin.py**: aqui podemos registrar nossos modelos, os quais serão usados pelo Django para criar a interface gráfica de admin, para gerenciamento da aplicação.
- **apps.py**: configurações da aplicação.
- **models.py**: aqui especificamos as nossas classes de modelo. É onde fazemos o mapeamento objeto relacional.
- **tests.py**: funções para testes unitários
- **views.py**: aqui definimos nossas views, ou seja, classes Python que recebem requisições, interagem com o modelo e devolvem respostas. É o V da arquitetura MTV do Django, semelhante aos controllers do MVC.
- pasta **migrations**: é um pacote/módulo Python, pois internamente possui um arquivo `__init__.py`. Quando executado, faz a migração de nossa base acontecer. Em geral, quando atualizamos nossas classes de modelo, novos arquivos serão incluídos neste diretório, representando o código necessário para deixar o modelo do banco de dados de acordo com as classes de modelo.

O arquivo **settings.py** (dentro da pasta do projeto) possui uma variável chamada **INSTALLED_APPS**, associada a uma lista Python que contém os nomes das aplicações que fazem parte deste projeto.



```

settings.py - primeiro_projeto - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
PRIMEIRO_PROJETO
  primeira_aplicacao
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  primeiro_projeto
    __pycache__
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
    db.sqlite3
    manage.py
settings.py
primeiro_projeto > settings.py > ...
production!
26 DEBUG = True
27
28 ALLOWED_HOSTS = []
29
30
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
42 MIDDLEWARE = [
43     "django.middleware.security.SecurityMiddleware",
44     "django.contrib.sessions.middleware.SessionMiddleware",
45     "django.middleware.common.CommonMiddleware",
46     "django.middleware.csrf.CsrfViewMiddleware",
47     "django.contrib.auth.middleware.AuthenticationMiddleware",
48     "django.contrib.messages.middleware.MessageMiddleware",
49     "django.middleware.clickjacking.XFrameOptionsMiddleware"
]

```

Vamos incluir o nome de nossa aplicação ali.

```
...
ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "primeira_aplicacao"
]

MIDDLEWARE = [
...
]
```

No terminal em que o servidor está em execução, observe que um **Hot Reload** acontece.

```
/home/rodrigo/workspaces_insp5502/pessoal/django/primeiro_projeto/primeiro_projeto/settings.py changed, reloading.
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 10, 2023 - 22:25:08
Django version 4.2.5, using settings 'primeiro_projeto.settings'.
Starting development server at http://127.0.0.1:8080/
Quit the server with CONTROL-C.
```

Tente também digitar um nome errado e verifique que o servidor passa a falhar.

```
...
ALLOWED_HOSTS = []

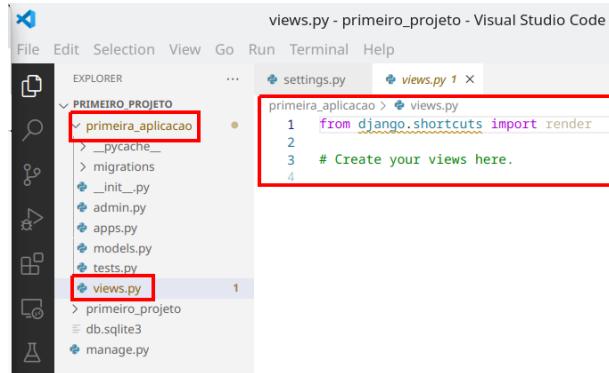
# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "nome_errado"
]

MIDDLEWARE = [
```

Depois disso, ajuste o nome da aplicação, deixando o correto.

2.4 (Criando uma View “Hello, World”) Lembre-se que uma View é responsável por receber requisições, eventualmente interagir com o modelo e devolver uma resposta ao cliente. Neste passo, vamos escrever uma view simples que faz um “Hello, World” com Django acontecer. Comece abrindo o arquivo **views.py**, dentro do diretório da aplicação.



A nossa view vai receber uma requisição HTTP e, por consequência, devolver uma resposta HTTP. Por isso, vamos importar o módulo `HttpResponse`.

```
from django.shortcuts import render
from django.http import HttpResponse
```

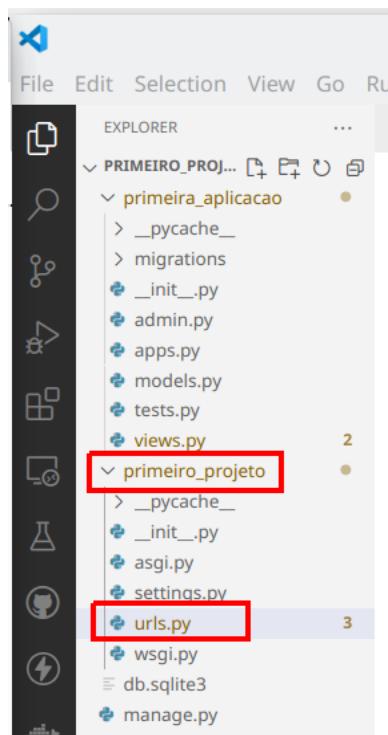
Agora, vamos escrever uma função chamada **index**. Ela recebe um objeto que representa a requisição e devolve um objeto do tipo `HttpResponse`, construído com o texto “Hello, Django”.

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello, Django')
```

Nota. Os nomes `index` e `request` são convenções muito comuns. Porém, não são obrigatórios.

O próximo passo é explicar a forma como essa função pode ser acionada. Qual URL precisa ser acessada para que ela entre em execução? Podemos fazer isso no arquivo **urls.py do projeto**.



Neste arquivo, vamos importar o módulo `views` de nossa aplicação. A seguir, vamos dizer que a função `index` dele deve entrar em execução quando uma requisição direcionada à raiz da aplicação for realizada.

```
...
"""
from django.contrib import admin
from django.urls import path
from primeira_aplicacao import views

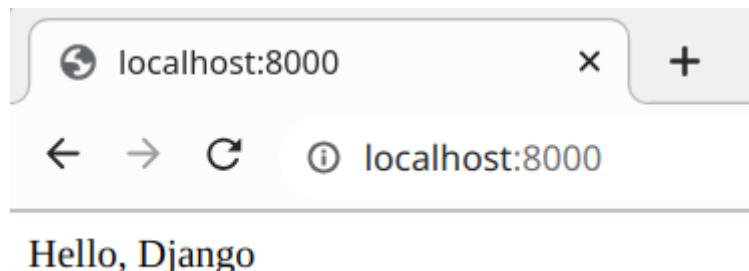
urlpatterns = [
    path("admin/", admin.site.urls),
    #cadeia vazia indica raiz da aplicação
    #chamamos a função index do módulo views
    #damos um nome (index) a este mapeamento para que possamos fazer
    #referência a ele no código posteriormente, se necessário
    path('', views.index, name="index"),
]
```

Conforme editamos os arquivos, o **Hot Reload** do servidor deve se encarregar de tornar disponíveis as novas funcionalidades, em geral, sem que seja necessário reiniciar o servidor.

Em seu navegador, visite novamente

localhost:8000

para obter algo como



Nota. A função path foi introduzida na versão 2 do Django. Antes de sua existência, o mapeamento era realizado com a função url. Ela ainda existe e funciona baseando-se em expressões regulares. Há também uma função chamada `re_path`, que opera como a path mas que admite o uso de expressões regulares. Se estiver curioso, descubra mais em

<https://docs.djangoproject.com/en/4.2/ref/urls/>

2.5 (Isolando as URLs da aplicação e incluindo elas no projeto com a função include)

Temos um único projeto que pode ser constituído por múltiplas aplicações e, no momento, temos um único arquivo `urls.py` de projeto, no qual especificamos as URLs de todas as aplicações dele. Uma das vantagens de termos um projeto com múltiplas aplicações é a reusabilidade. Uma aplicação deve ser um componente reutilizável, que pode ser desplulado de um projeto e plugado em outro. Assim, as suas URLs não podem ser definidas num arquivo de projeto. Idealmente, elas são definidas num arquivo próprio da aplicação e, quando a aplicação for incluída em um projeto, apenas indicamos a existência deste arquivo de alguma forma no projeto. Para isso, vamos usar a função `include`.

No arquivo `urls.py` do projeto, vamos importar a função `include` e, a seguir, explicar que requisições direcionadas a `host/primeira_aplicacao` devem ser resolvidas por funções escritas num arquivo de urls específico da aplicação chamada `primeira_aplicacao`.

urls.py - primeiro_projeto - Visual Studio Code

File Edit Selection View Go Run Terminal Help

EXPLORER

PRIMEIRO_PROJETO

- primeira_aplicacao
- _pycache_
- migrations
- __init__.py
- admin.py
- apps.py
- models.py
- tests.py
- views.py
- primeiro_projeto
- _pycache_
- __init__.py
- asgi.py
- settings.py
- urls.py
- wsgi.py
- db.sqlite3
- manage.py

primeiro_projeto > urls.py > ...

```

12     2. Add a URL to urlpatterns: path('', Home.as_view(),
13     name='home')
14     Including another URLconf
15     1. Import the include() function: from django.urls
16     import include, path
17     2. Add a URL to urlpatterns: path('blog/', include
18     ('blog.urls'))
19     """
20
21     from django.contrib import admin
22     from django.urls import path
23     from primeira_aplicacao import views
24
25     urlpatterns = [
26         path("admin/", admin.site.urls),
27         #cadeia vazia indica raiz da aplicação
28         #chamamos a função index do módulo views
29         #damos um nome (index) a este mapeamento para que
30         #possamos fazer referência a ele no código
31         #posteriormente, se necessário
32         path('', views.index, name="index"),
33     ]
34
35     """

```

```

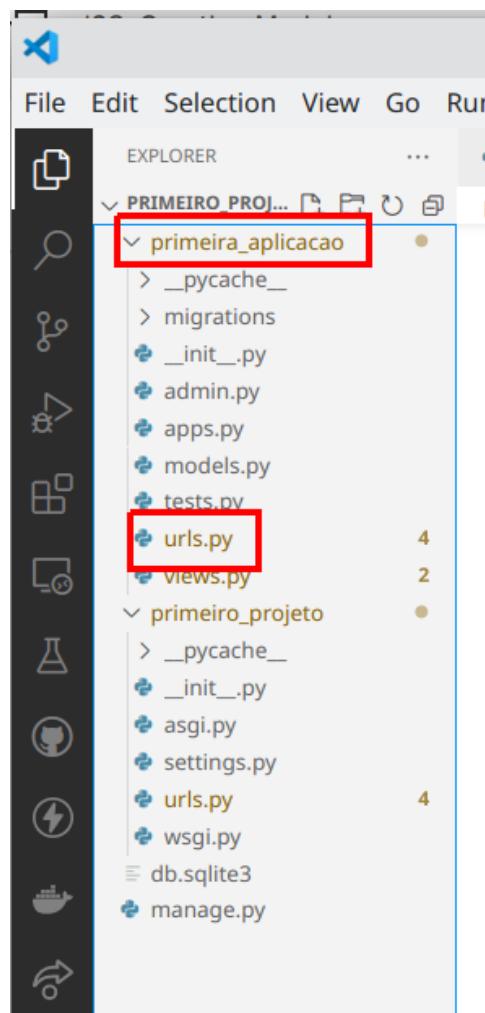
"""
from django.contrib import admin
from django.urls import path
from django.conf.urls import include
from primeira_aplicacao import views

urlpatterns = [
    path("admin/", admin.site.urls),
    #cadeia vazia indica raiz da aplicação
    #chamamos a função index do módulo views
    #damos um nome (index) a este mapeamento para que possamos fazer
    #referência a ele no código posteriormente, se necessário
    path('', views.index, name="index"),
    path('primeira_aplicacao/', include('primeira_aplicacao.urls'))
]
...

```

Cuidado. É fundamental a existência da / depois do nome de interesse. Ou seja, "primeira_aplicacao/" funciona. "primeira_aplicacao" não funciona.

Observe que estamos “incluso” o módulo urls da primeira_aplicacao. Um módulo é um simples arquivo .py. Entretanto, não há um arquivo chamado urls.py no diretório primeira_aplicacao. Por isso, vamos criar um.



Neste arquivo, precisamos especificar uma variável chamada **urlpatterns** (o nome precisa ser esse mesmo) e ela deve referenciar uma lista. A lista contém os mapeamentos desejados, como antes. Neste caso, vamos dizer que requisições direcionadas à raiz devem ser atendidas pela função index.

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

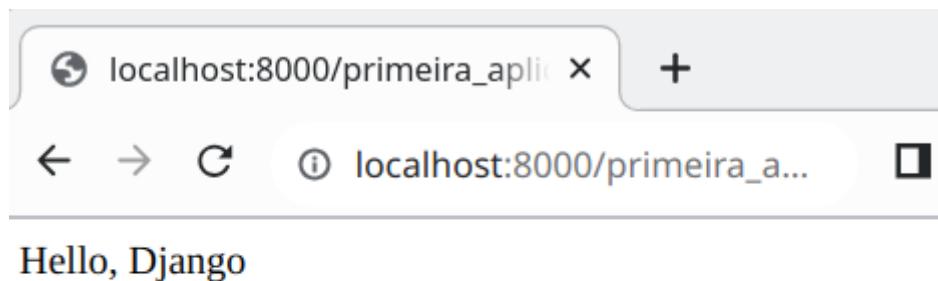
urlpatterns = [
    path('', views.index, name="index")
]
```

Observe que “raiz”, neste caso, significa host:porta/primeira_aplicacao/. Essa é a raiz desta aplicação. A raiz host:porta/ continua sendo do projeto e a função a ser acionada mediante este acesso permanece sendo definida no arquivo **urls.py** do projeto. Por acaso, ela ainda é a função index desta aplicação.

No seu navegador, acesse

localhost:8000/primeira_aplicacao

Veja o resultado esperado.



Observe que o padrão localhost:8000/primeira_aplicacao foi utilizado por simplicidade. Podemos escolher qualquer padrão de acesso para nossa aplicação. No arquivo **urls.py do projeto**, faça o seguinte ajuste.

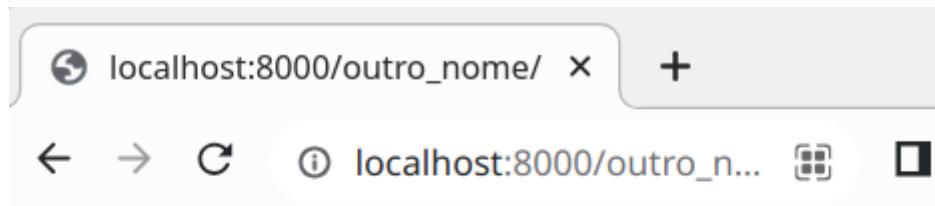
```
...
"""
from django.contrib import admin
from django.urls import path
from django.conf.urls import include
from primeira_aplicacao import views

urlpatterns = [
    path("admin/", admin.site.urls),
    #cadeia vazia indica raiz da aplicação
    #chamamos a função index do módulo views
    #damos um nome (index) a este mapeamento para que possamos fazer
    #referência a ele no código posteriormente, se necessário
    path('', views.index, name="index"),
    path('outro_nome/', include('primeira_aplicacao.urls'))
]
```

Em seu navegador, acesse

localhost:8000/outro_nome

e observe o resultado.



Volte ao normal, mantendo o texto **primeira_aplicacao**.

```

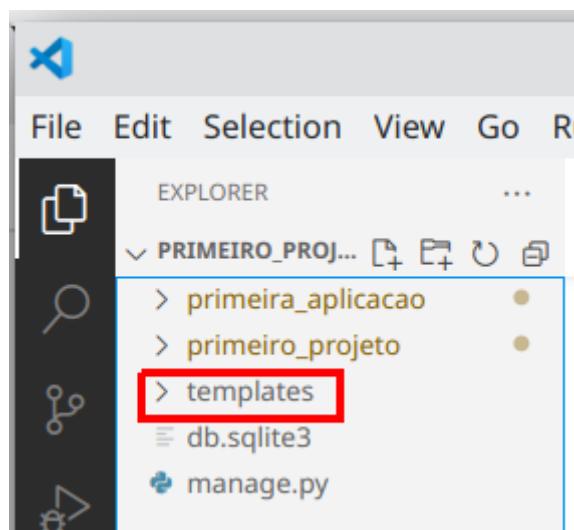
...
"""

from django.contrib import admin
from django.urls import path
from django.conf.urls import include
from primeira_aplicacao import views

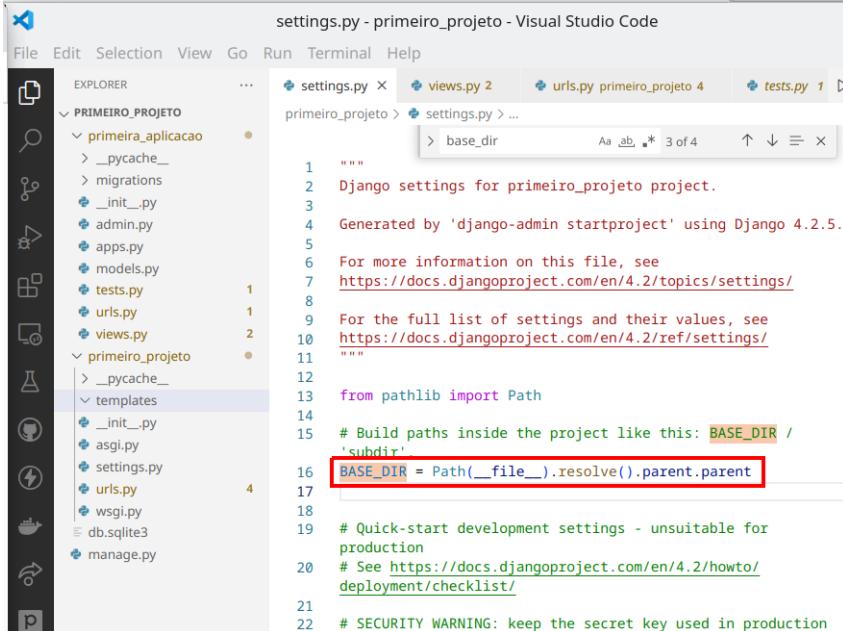
urlpatterns = [
    path("admin/", admin.site.urls),
    #cadeia vazia indica raiz da aplicação
    #chamamos a função index do módulo views
    #damos um nome (index) a este mapeamento para que possamos fazer
    #referência a ele no código posteriormente, se necessário
    path('', views.index, name="index"),
    path('primeira_aplicacao/', include('primeira_aplicacao.urls'))
]
...

```

2.6 (Produção de HTML do lado do servidor com templates) Django oferece tags de template, as quais nos permitem gerar código HTML do lado do servidor. O primeiro passo para utilizar templates, é **criar um diretório chamado templates na raiz do projeto**. Observe que ele fica fora do diretório da aplicação e fora do diretório interno que leva o nome do projeto, onde se encontram os arquivos de configuração.



Depois disso, precisamos dizer ao Django que esse diretório existe e que ele contém arquivos de template. Para tal, abra o arquivo **settings.py** do projeto. Observe que este arquivo possui uma variável chamada **BASE_DIR**. Ele representa a raiz do projeto. Por ter sido construído como um objeto Path, a sua representação é independente de sistema operacional. Apenas observe.



```

settings.py - primeiro_projeto - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ... settings.py x views.py 2 urls.py primeiro_projeto 4 tests.py 1
primeiro_projeto > settings.py > ...
> base_dir Aa ab 3 of 4
1 """
2 Django settings for primeiro_projeto project.
3
4 Generated by 'django-admin startproject' using Django 4.2.5.
5
6 For more information on this file, see
7 https://docs.djangoproject.com/en/4.2/topics/settings/
8
9 For the full list of settings and their values, see
10 https://docs.djangoproject.com/en/4.2/ref/settings/
11 """
12
13 from pathlib import Path
14
15 # Build paths inside the project like this: BASE_DIR /
16 # 'subdir'.
16 BASE_DIR = Path(__file__).resolve().parent.parent
17
18 # Quick-start development settings - unsuitable for
19 # production
20 # See https://docs.djangoproject.com/en/4.2/howto/deployment/checklist/
21
22 # SECURITY WARNING: keep the secret key used in production

```

Vamos definir uma variável chamada **TEMPLATE_DIR** (nome arbitrário) da seguinte forma.

```

...
ROOT_URLCONF = "primeiro_projeto.urls"

#operador / sobrecregido para objetos Path
#representa um separador independente de SO
TEMPLATE_DIR = BASE_DIR / Path('templates')
TEMPLATES = [
...

```

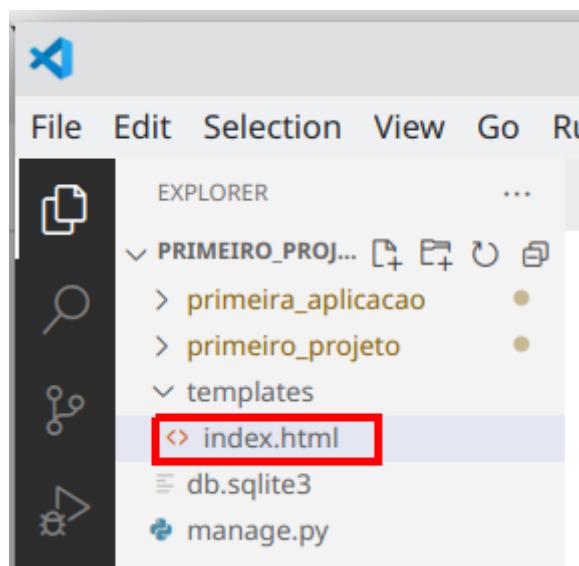
Ela referencia um objeto Path que representa o diretório de templates de maneira independente de sistema operacional.

O próximo passo é incluir este objeto na lista “DIRS” de TEMPLATES.

```
...
ROOT_URLCONF = "primeiro_projeto.urls"

#operador / sobreescarregado para objetos Path
#representa um separador independente de SO
TEMPLATE_DIR = BASE_DIR / Path('templates')
TEMPLATES = [
{
    "BACKEND": "django.template.backends.django.DjangoTemplates",
    "DIRS": [TEMPLATE_DIR],
    "APP_DIRS": True,
    "OPTIONS": {
        ...
    }
}
```

A seguir, criamos um arquivo chamado **index.html** na pasta templates.



Veja seu conteúdo inicial.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
                                         initial-scale=1.0">
    <title>Primeira aplicação</title>
  </head>
  <body>
    <h1>Estamos no arquivo index.html!</h1>
  </body>
</html>

```

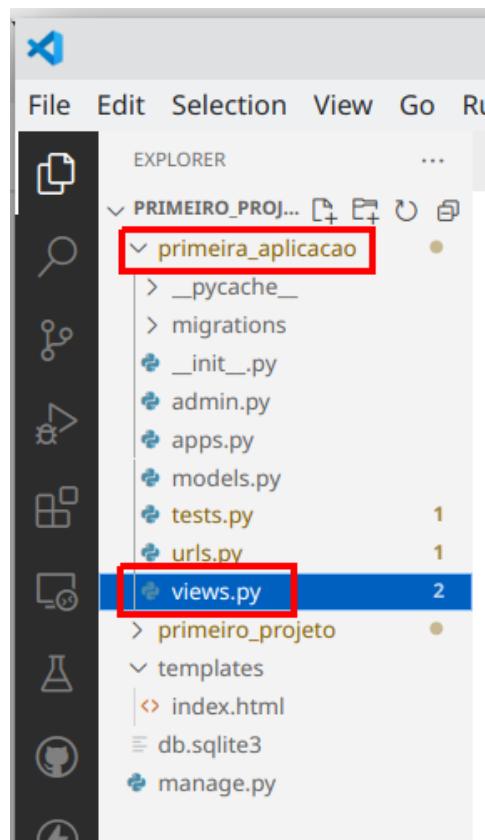
Podemos definir **variáveis de template Django** e a elas associar valores arbitrários. Depois disso, utilizamos o operador de interpolação ({{ }}) para fazer a substituição da variável pelo seu valor associado, no contexto em que ela aparece. Veja este exemplo.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
                                         initial-scale=1.0">
    <title>Primeira aplicação</title>
  </head>
  <body>
    <h1>Estamos no arquivo index.html!</h1>
    <p>{{minha_primeira_variavel}}</p>
  </body>
</html>

```

Resta definir a variável. Vamos ao arquivo **views.py** da aplicação para tal.



Neste arquivo, vamos utilizar a própria função **index** para fazer a definição das variáveis. Elas são colocadas num dicionário Python. A função render recebe

- a requisição HTTP
- o nome do arquivo html ao qual direcionar a requisição
- a coleção de variáveis

Observe que a função render já havia sido importada previamente.

Nota. A palavra **render** tem diferentes significados dependendo do contexto em que é utilizada. De modo geral, podemos entender que renderizar significa **produzir algo em função de matéria prima bruta**. Ou seja, “**tornar pronto**”. No contexto em que estamos, **renderizar** significa **produzir o arquivo HTML que será entregue ao cliente**, em função das diferentes matérias primas envolvidas, como as variáveis, conteúdos vindos do banco de dados etc.

```

from django.shortcuts import render
from django.http import HttpResponse

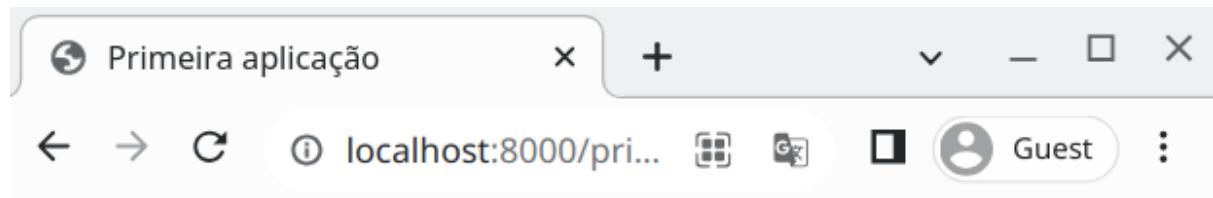
def index(request):
    variaveis = {
        'minha_primeira_variavel': "Hello, variáveis!!"
    }
    return render(request, 'index.html', context=variaveis)

```

Em seu navegador, acesse

http://localhost:8000/primeira_aplicacao/

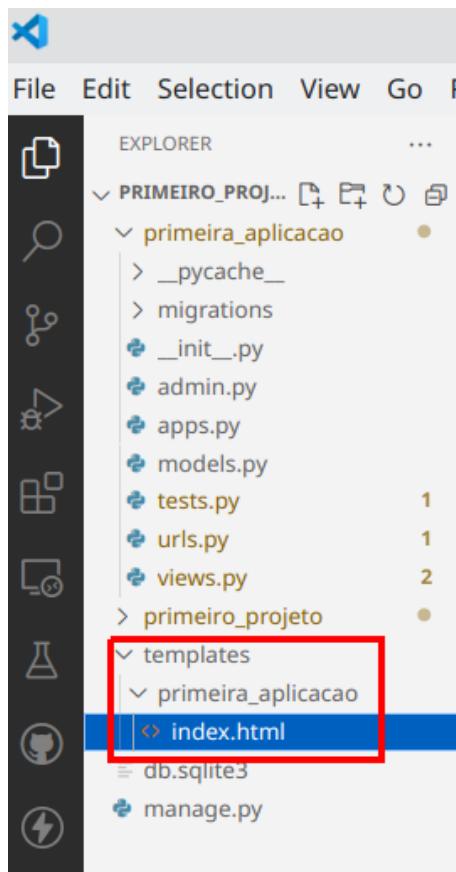
para obter o seguinte resultado. Se necessário, reinicie o servidor.



Estamos no arquivo `index.html`!

Hello, variáveis!!

Pode ser interessante separar os templates de aplicações diferentes. É uma prática comum **criar um subdiretório dentro do diretório templates** com o nome de cada aplicação. Façamos isso para a aplicação `primeira_aplicacao`. Depois disso, **arrastamos o arquivo `index.html` para dentro desse novo diretório**.



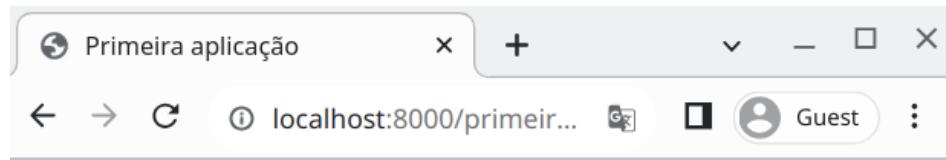
No arquivo **views.py** da aplicação, ajustamos a referência para que o novo diretório seja considerado. Ajuste também o texto associado à variável.

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

def index(request):
    variaveis = {
        'minha_primeira_variavel': "Hello, variáveis depois de alterar o
        diretório!"
    }
    return render(request, 'primeira_aplicacao/index.html', context=variaveis)
```

No seu navegador, faça um novo teste.

http://localhost:8000/primeira_aplicacao/

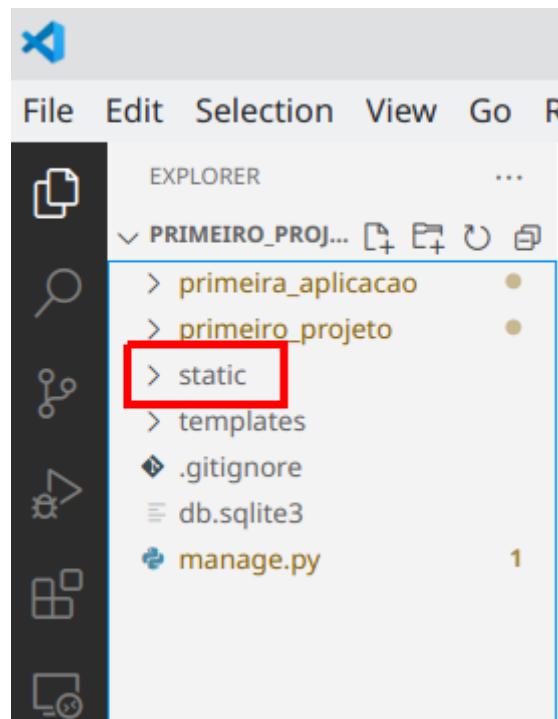


Estamos no arquivo index.html!

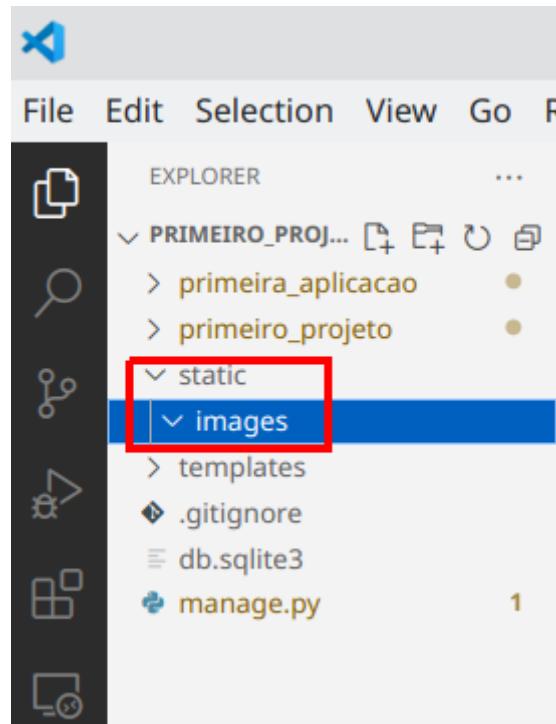
Hello, variáveis depois de alterar o diretório!

2.7 (Lidando com arquivos estáticos (como figuras)) Nesta parte, veremos como podemos incluir figuras em páginas HTML geradas pelo Django. A ideia é fazermos as configurações adequadas para que uma figura possa ser acessada de uma forma parecida como fizemos com texto: associamos a figura a uma variável de template e usamos a variável de template no arquivo HTML.

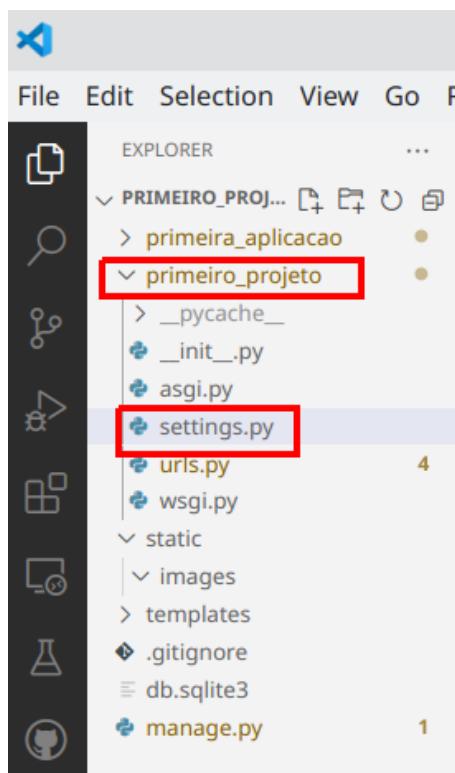
Comece criando um diretório chamado **static** na raiz do projeto, fora da pasta da aplicação e fora da pasta que leva o nome do projeto, aquela que contém os arquivos de configuração. Ela fica no mesmo nível da pasta de templates que criamos antes.



Temos diversos tipos de arquivos estáticos, como arquivos .css e .js também. Assim, **vamos criar uma pasta própria para o armazenamento de imagens, chamada images, subpasta de static.**



Tal qual fizemos com o diretório de templates, vamos usar o arquivo **settings.py** do projeto para explicar onde se encontram os arquivos estáticos da aplicação.



```

...
USE_I18N = True

USE_TZ = True

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.2/howto/static-files/


STATIC_DIR = BASE_DIR / Path('static')
STATIC_URL = "static/"

# Default primary key field type
#
# https://docs.djangoproject.com/en/4.2/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"

...

```

Observe que já existe uma variável chamada STATIC_URL. Como seu nome no singular sugere, ela aponta para um único diretório. Ocorre que podemos estar interessados em especificar uma coleção de diretórios, um para cada aplicação diferente, tal como fizemos com os templates. Por isso, vamos criar uma variável chamada **STATICFILES_DIRS** (o nome tem que ser exatamente esse) e a ela associar uma lista contendo o nosso novo diretório.

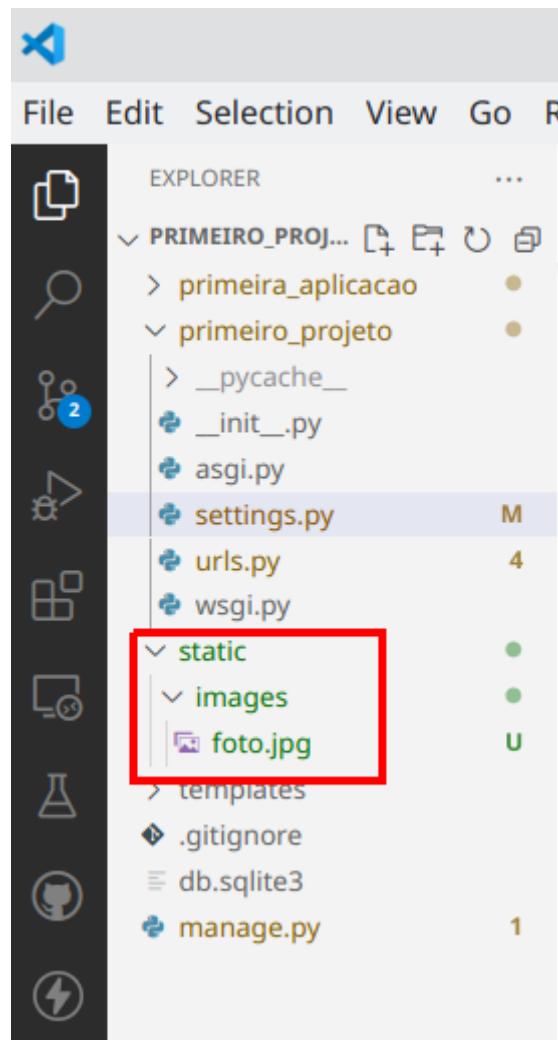
```

...
STATIC_DIR = BASE_DIR / Path('static')
STATICFILES_DIRS = [
    STATIC_DIR
]
STATIC_URL = "static/"
...
```

Visite o site Pexels e faça o download de uma foto qualquer.

www.pexels.com

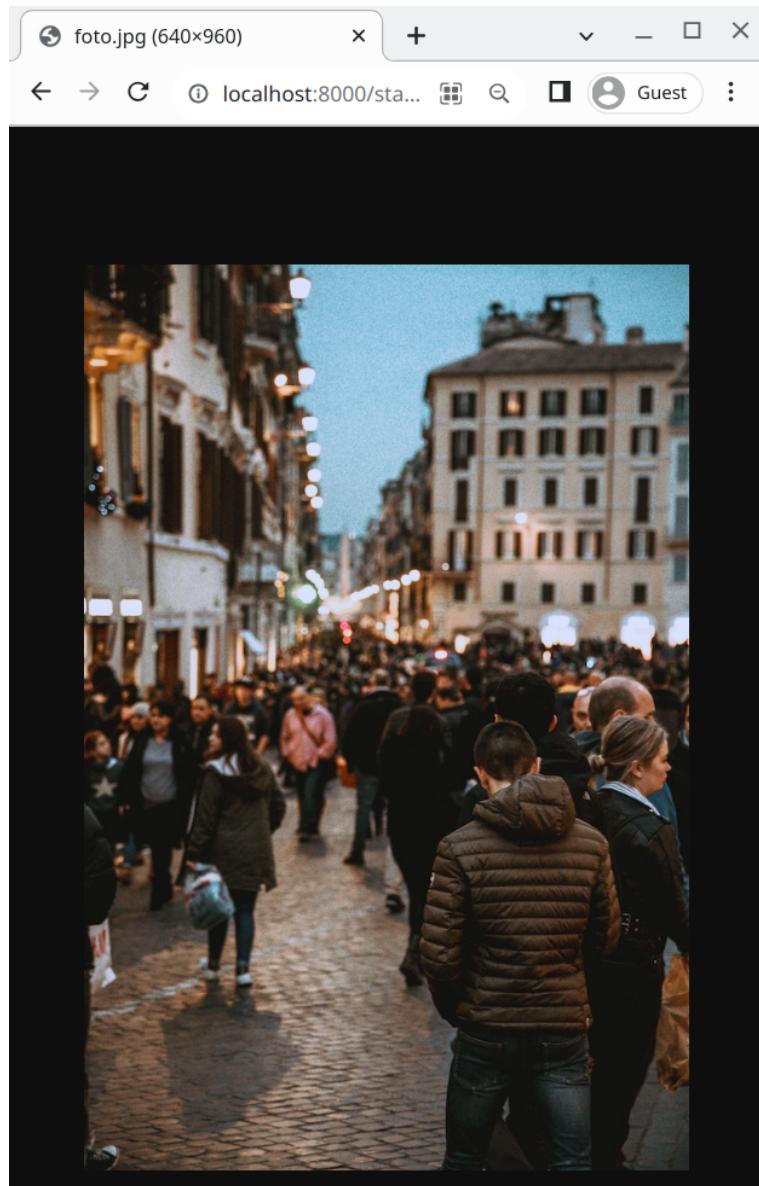
Feito o download, armazene a foto em sua pasta static/images. Se desejar, renomeie a foto para simplificar. Neste exemplo, seu nome passou a ser **foto.jpg**.



Observe que os recursos estáticos, por padrão, podem ser acessados diretamente. No seu navegador, visite

<http://localhost:8000/static/images/foto.jpg>

e obtenha o seguinte resultado. Se necessário, reinicie o servidor.



Claro, talvez não seja de interesse permitir que todos os arquivos estáticos da aplicação sejam acessados diretamente assim. No futuro veremos como prevenir isso.

Agora queremos incluir essa figura na página HTML que possuímos. No arquivo **templates/primeira_aplicacao/index.html**, o primeiro passo é utilizar uma instrução própria do Django que faz com que os recursos estáticos se tornem acessíveis na página.

```
<!DOCTYPE html>
{ % load static %}
<html lang="en">
<head>
...
```

Agora podemos acessar arquivos estáticos nesta página HTML. Para acessar uma imagem, vamos usar a tag img regular do HTML, porém usando uma tag de template do Django como src.

```
<!DOCTYPE html>
{ % load static %}
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Primeira aplicação</title>
</head>
<body>
<h1>Estamos no arquivo index.html!</h1>
<p>{{minha_primeira_variavel}}</p>
<p>Veja uma imagem:</p>

</body>
</html>
```

No seu navegador, acesse

<http://localhost:8000/>

ou

http://localhost:8000/primeira_aplicacao/

e obtenha o seguinte resultado.

Primeira ap! x view-source x + v - □ ×

localhost:800... Guest :

Estamos no arquivo index.html!

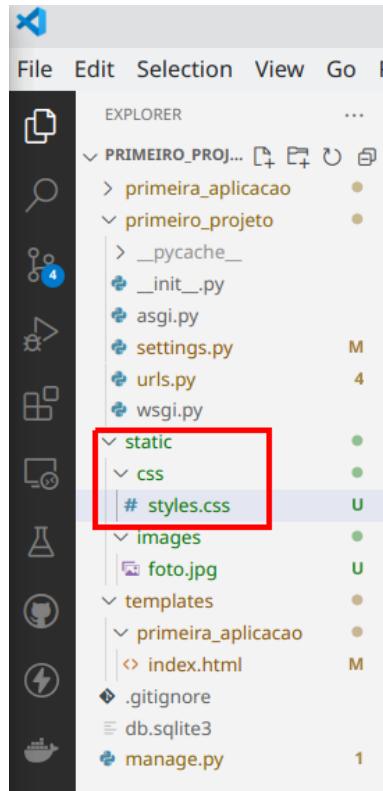
Hello, variáveis depois de alterar o diretório!

Veja uma imagem:



Nota. As duas URLs podem ser acessadas pois ainda temos a mesma view mapeada para atendimento nos dois padrões.

Claro, também podemos incluir arquivos css. Comece criando uma pasta chamada **css**, subpasta de **static**. Crie também um arquivo chamado **styles.css**.



Veja o conteúdo do arquivo **styles.css**. Um seletor simples com uma única regra que tem impacto sobre todos os parágrafos da página.

```
p {  
    color: blue;  
}
```

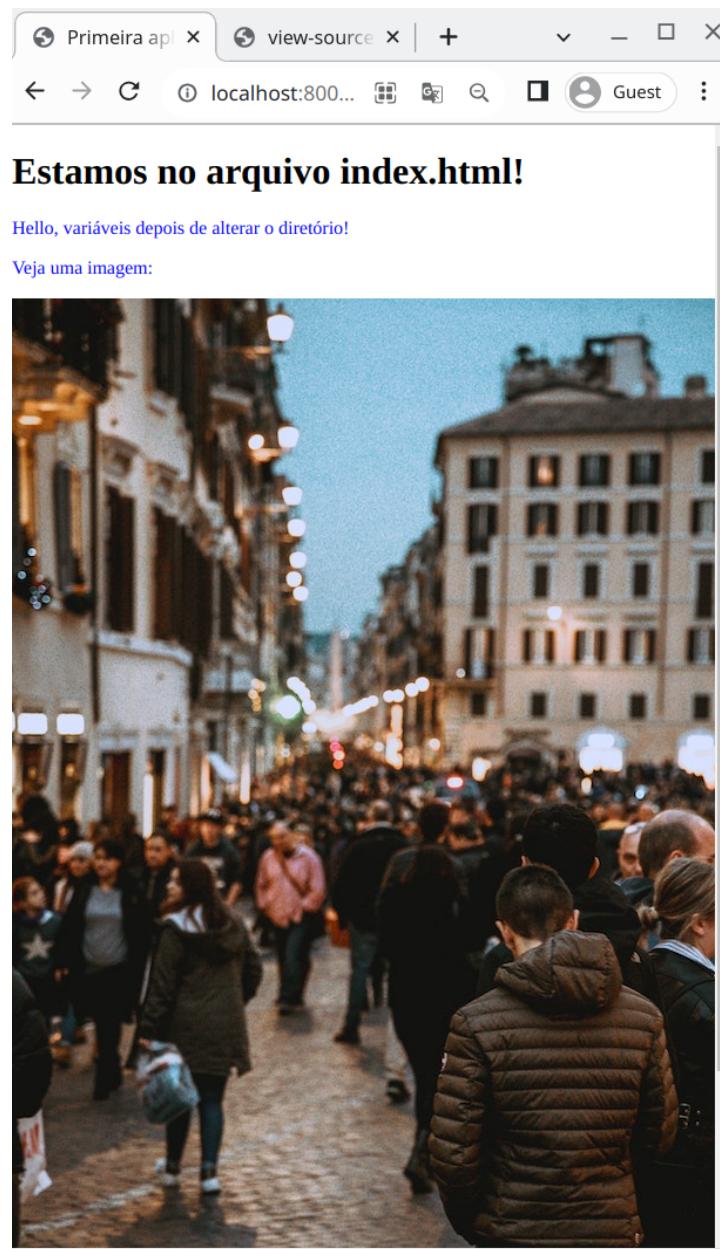
No arquivo **index.html**, precisamos importar o arquivo css.

```
<!DOCTYPE html>
{ % load static %}
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="{% static 'css/styles.css' %}">
  <title>Primeira aplicação</title>
</head>
...
```

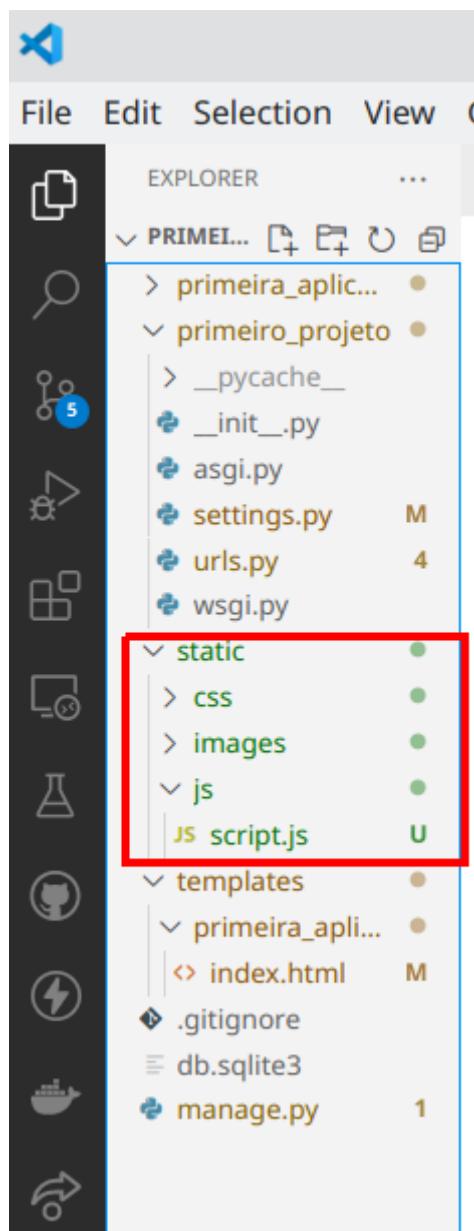
Acesse a página novamente no navegador.

http://localhost:8000/primeira_aplicacao/

Veja o resultado.



Faça também um teste com Javascript. Crie uma pasta chamada **js**, subpasta de static. Nela, crie um arquivo chamado **script.js**.



Veja o conteúdo do arquivo **script.js**.

```
function hey () {  
  alert('hey')  
}
```

No arquivo **index.js**

- defina um botão e vincule a função `hey` ao seu evento `onclick`
- importe o arquivo `script.js` usando uma tag de template do django

```
<!DOCTYPE html>
{%
  load static %
}
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
    <title>Primeira aplicação</title>
  </head>
  <body>
    <h1>Estamos no arquivo index.html!</h1>
    <button onclick="hey()">Hey</button>
    <p>{{minha_primeira_variavel}}</p>
    <p>Veja uma imagem:</p>
    
    <script src="{% static 'js/script.js' %}"></script>
  </body>
</html>
```