

.

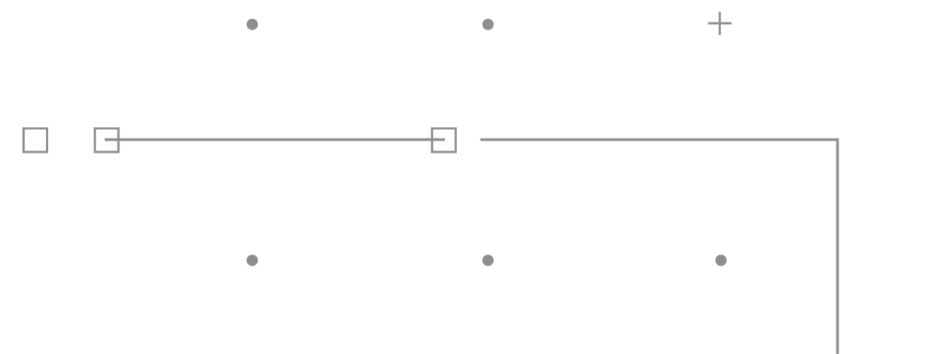
F

-

I

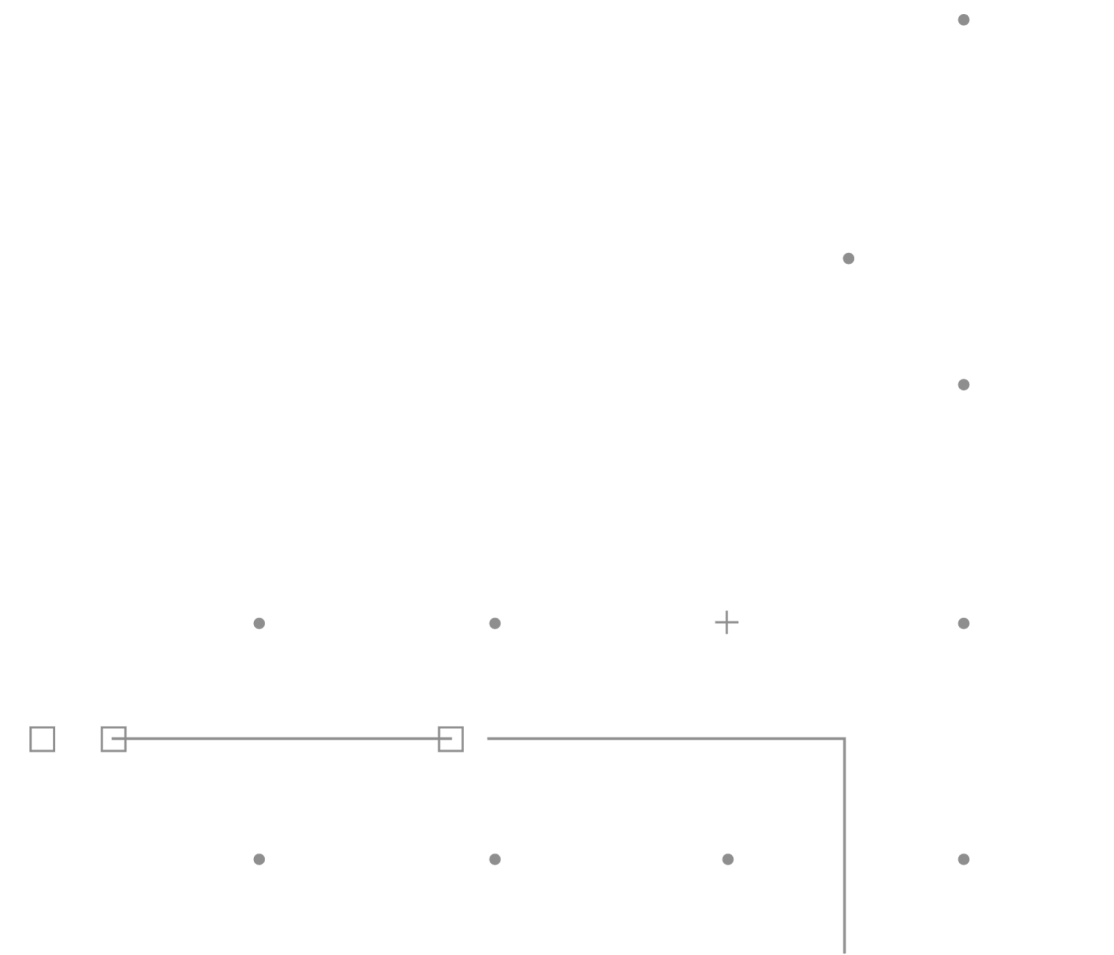
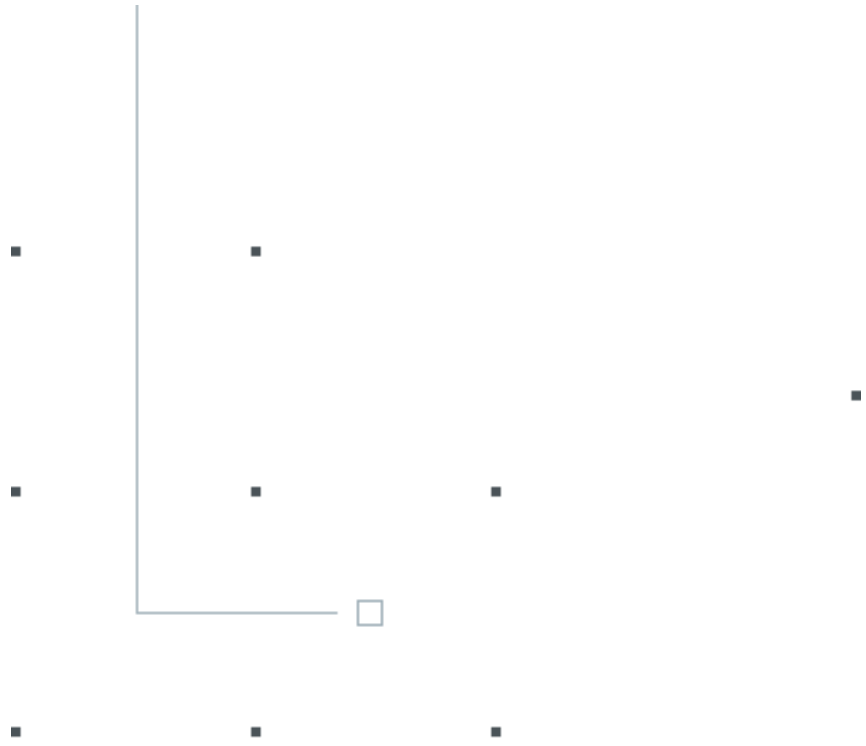
A

P



# Virtualenv

## Django






# AMBIENTE

Quando se trabalha com Python, é muito comum ouvir falar de **virtualenv**.

**Virtualenv**, ou virtual environment, nada mais é que um ambiente python que funciona de forma isolada do sistema operacional. Ou seja, todos os pacotes contidos em um virtualenv não estão disponíveis no sistema operacional e vice-versa.





# AMBIENTE

Para criar um virtualenv, basta rodar o comando:

```
python3 -m venv {nome do virtualenv}
```

Exemplo:

```
python3 -m venv my_env
```



# AMBIENTE

E como ele funciona?

Ao criar um ambiente virtual, a seguinte estrutura será criada:

```
venv/
├── bin/
│   ├── Activate.ps1
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── pip
│   ├── pip3
│   ├── pip3.10
│   ├── python
│   ├── python3
│   └── python3.10
├── include/
├── lib/
│   └── python3.10/
│       └── site-packages/
├── lib64/
│   └── python3.10/
│       └── site-packages/
└── pyvenv.cfg
```

# AMBIENTE

Depois de criado, para ativá-lo, basta rodar o comando:

```
my_env\bin\Scripts\activate
```

Usuários de plataforma Unix (Linux/MacOS):

```
source my_env/bin/activate
```

```
bash-3.2$ python3 -m venv my_env
bash-3.2$ source my_env/bin/activate
(my_env) bash-3.2$
```

# AMBIENTE

Ao ativar um ambiente virtual, toda vez que você rodar o comando `python`, será usado o `python` que existe dentro dele

Veja com o ambiente virtual fica o caminho do `python`

```
(.venv) → ~ which python3  
/home/leonardo/.venv/bin/python  
(.venv) → ~
```

E fora do ambiente virtual

```
→ ~ which python3  
/usr/bin/python3  
→ ~
```

# AMBIENTE

Para desativar, basta executar o comando:

**deactivate**

```
[bash-3.2$ python3 -m venv my_env  
[bash-3.2$ source my_env/bin/activate  
[my_env] bash-3.2$ deactivate  
bash-3.2$
```



# AMBIENTE

Para instalar pacotes, basta rodar o comando

```
pip install <nome_pacote>
```

Exemplo:

```
pip install requests
```

```
(my_env) → ~ pip install requests
Collecting requests
  Downloading requests-2.28.2-py3-none-any.whl (62 kB)
    _____ 62.8/62.8 KB 2.9 MB/s eta 0:00:00
Collecting charset-normalizer<4,>=2
  Downloading charset_normalizer-3.0.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (198 kB)
    _____ 198.8/198.8 KB 13.6 MB/s eta 0:00:00
Collecting certifi>=2017.4.17
  Downloading certifi-2022.12.7-py3-none-any.whl (155 kB)
    _____ 155.3/155.3 KB 24.8 MB/s eta 0:00:00
Collecting idna<4,>=2.5
  Downloading idna-3.4-py3-none-any.whl (61 kB)
    _____ 61.5/61.5 KB 9.5 MB/s eta 0:00:00
Collecting urllib3<1.27,>=1.21.1
  Downloading urllib3-1.26.14-py2.py3-none-any.whl (140 kB)
    _____ 140.6/140.6 KB 20.7 MB/s eta 0:00:00
Installing collected packages: charset-normalizer, urllib3, idna, certifi, requests
Successfully installed certifi-2022.12.7 charset-normalizer-3.0.1 idna-3.4 requests-2.28.2 urllib3-1.26.14
(my_env) → ~
```

# AMBIENTE

Para visualizar os pacotes instalado, basta rodar o comando:

```
pip list
```

```
(my_env) → ~ pip list
Package            Version
-----
certifi             2022.12.7
charset-normalizer  3.0.1
idna                3.4
pip                 22.0.2
requests            2.28.2
setuptools          59.6.0
urllib3             1.26.14
(my_env) → ~
```

# AMBIENTE - Por que utilizar?

Alguns dos principais motivos para se utilizar um virtualenv são:

- Ambientes como Linux e MacOS já vem com o Python instalado por padrão e com isso, um monte de biblioteca. Criando seu virtualenv, você consegue ter um melhor controle sobre o que sua aplicação realmente precisa.
- Diferentes projetos com diferentes versões de bibliotecas.
- Evita problemas de permissão. Em ambientes corporativos nem sempre é possível instalar novas bibliotecas.

# DJANGO

Django é um framework que implementa o protocolo HTTP e conta com recursos muito robustos como autenticação, ORM, criptografia, etc.

Após criarmos nosso virtualenv, para instalar basta rodar o comando:

```
pip install Django
```

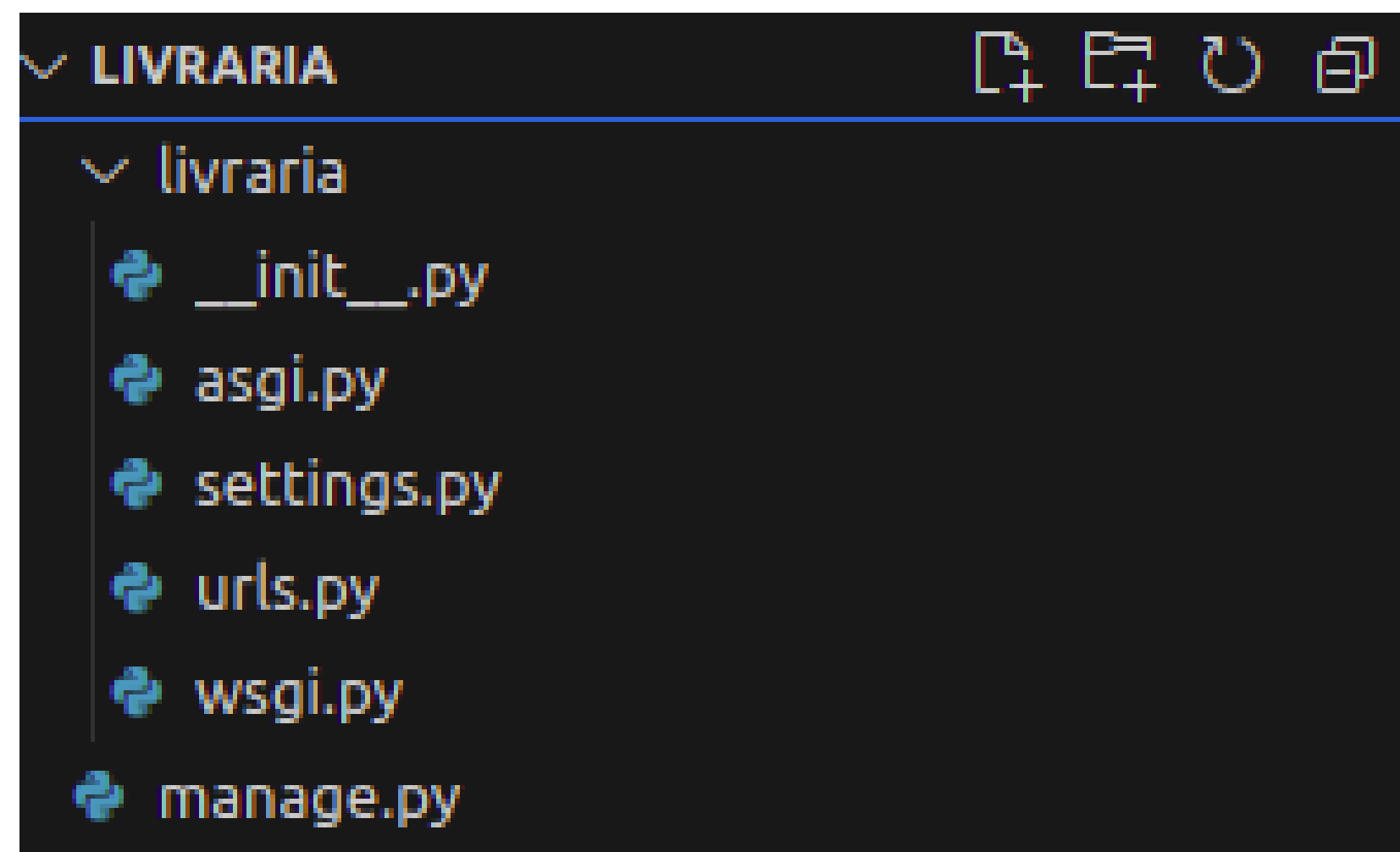
```
(.venv) → django pip install Django
Collecting Django
  Downloading Django-4.2.1-py3-none-any.whl (8.0 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.0/8.0 MB 16.6 MB/s eta 0:00:00
Collecting asgiref<4,>=3.6.0
  Downloading asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1
  Downloading sqlparse-0.4.4-py3-none-any.whl (41 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 41.2/41.2 KB 6.8 MB/s eta 0:00:00
Collecting typing-extensions>=4
  Downloading typing_extensions-4.6.3-py3-none-any.whl (31 kB)
Installing collected packages: typing-extensions, sqlparse, asgiref, Django
Successfully installed Django-4.2.1 asgiref-3.7.2 sqlparse-0.4.4 typing-extensions-4.6.3
(.venv) → django
```

# DJANGO

Para criar um projeto utilizando o Django, deve-se executar o seguinte comando:

```
django-admin startproject projeto
```

Onde projeto é o nome do seu projeto. Ao criar seu projeto, será criada uma pasta com a seguinte estrutura:



# DJANGO

Iremos criar um arquivo chamado **requirements.txt** na raiz do diretório. É comum criar esse arquivo para colocar quais as bibliotecas que sua aplicação utiliza e se precisar criar um novo ambiente, basta aproveitar este arquivo para instalar essas bibliotecas.

No conteúdo do arquivo, terá apenas uma linha:

```
requirements.txt
1  Django==4.2.1
2
```

# DJANGO

E para utilizarmos esse arquivo `requirements.txt`, com seu ambiente virtual ativado, basta executar:

```
pip install -r requirements.txt
```

Todas as bibliotecas listadas nesse arquivo irão ser instaladas. Caso elas já estejam instaladas, mas com uma versão diferente da especificada no `requirements.txt`, a mesma será removida e instalada a versão que estiver informada no `requirements.txt`

# DJANGO

Para ver sua aplicação rodando, basta rodar o seguinte comando:

```
python manage.py runserver
```

```
(django) → livraria 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.
June 02, 2023 - 13:05:43
Django version 4.2.1, using settings 'livraria.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```



# DJANGO

Acesse no browser a seguinte URL:

<http://localhost:8000>


django


View [release notes](#) for Django 4.2




The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

 **Django Documentation**  
Topics, references, & how-to's

 **Tutorial: A Polling App**  
Get started with Django

 **Django Community**  
Connect, get help, or contribute

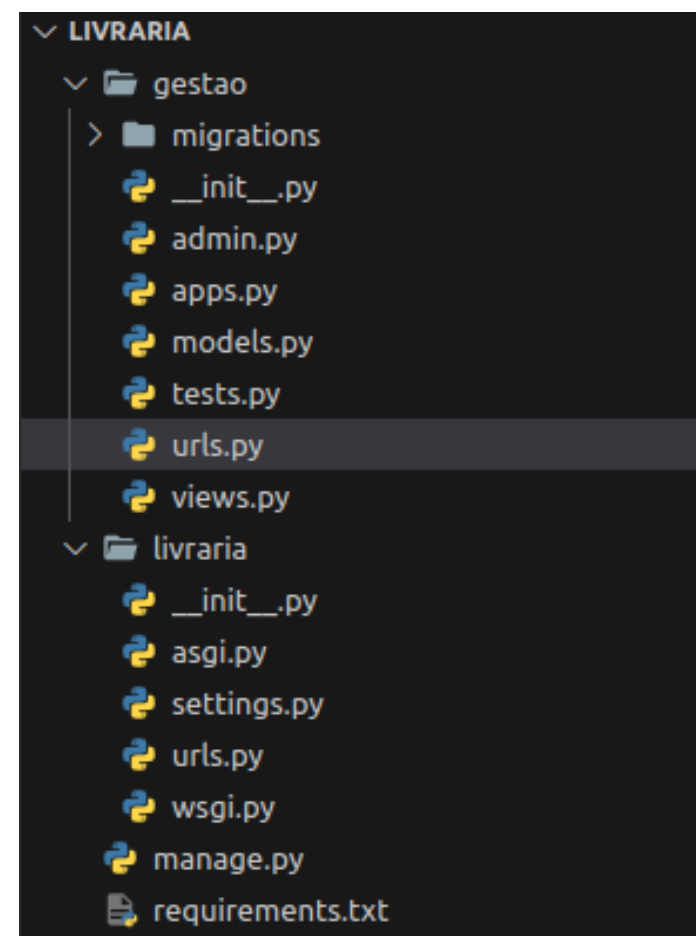
# DJANGO

O Django organiza o projeto através de aplicações dentro dele. Para criar uma aplicação, execute o comando:

```
python manage.py startapp aplicacao
```

Onde aplicação é o nome da `aplicacao` que deseja criar.

Foi criada uma aplicação chamada `gestao`. Seu projeto ficara com a seguinte estrutura:



# DJANGO

Considerando a estrutura citada anteriormente, existe o arquivo `views.py` dentro da pasta `gestao`.  
Dentro dele que iremos colocar como iremos tratar as requisições feitas pelos clientes.  
Deixe o arquivo com o seguinte conteúdo:

```
from django.http import HttpResponse

# Create your views here
def index(request):
    return HttpResponse("Hello, world")
```

# DJANGO

Ainda dentro da aplicação criada, no mesmo diretório onde está o `views.py`, crie um arquivo chamado `urls.py` com o seguinte conteúdo:

```
from django.urls import path

from . import views

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

# DJANGO

A função `path` que é importada no `urls.py`, serve para determinar um caminho que sua aplicação irá disponibilizar.

Então no exemplo do slide anterior, é criado um caminho raiz, que irá executar a função `index` dentro do módulo `views` e terá o apelido `index`, que poderá ser usado futuramente

# DJANGO

Agora na pasta do projeto, já existe um arquivo chamado `urls.py`. Edite esse arquivo para ter o seguinte conteúdo:

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

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

# DJANGO

Já a função `include` serve para importar um conjunto de URLs, dentro de outras URLs.

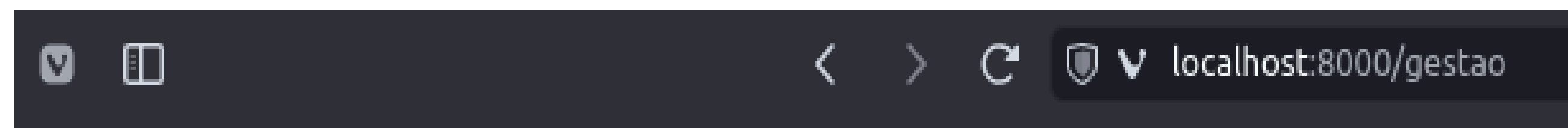
Como podemos ver, o caminho `/gestao` poderá ser acessado na aplicação e todas as URLs que tem dentro do `gestao.urls` serão incluídas como subcaminho.

Ou seja, se dentro do `gestao.urls` tivessem as URLs `/cadastrar` e `/editar`, por exemplo, com o `include` será possível acessar `/gestao/cadastrar` e `/gestao/editar`

# DJANGO

Rode novamente o comando para iniciar o servidor `python manage.py runserver`

Ao acessar a url <http://localhost:8000/gestao> irá aparecer o seguinte conteúdo:



Hello, world

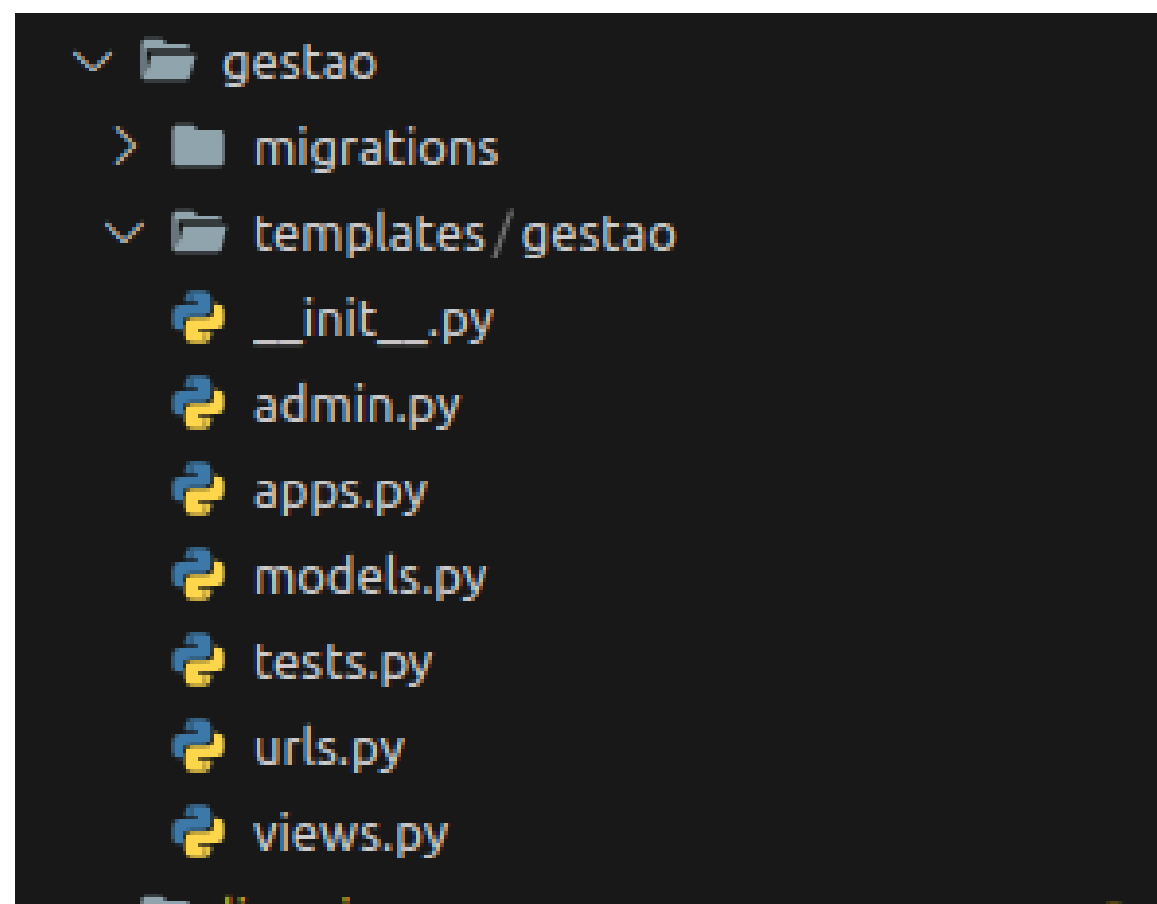


# Templates

Como podemos ver, é bem simples criar um endereço que será acessado pelo usuário da sua aplicação. Mas estamos retornando um texto básico.

Com o Django, é possível utilizar arquivos HTML. Os arquivos HTML no Django são conhecidos como `templates`

Para isso, dentro da pasta `gestao`, crie uma pasta chamada `templates` e dentro dela, crie uma nova chamada `gestao`.



# Templates

Dentro da pasta `templates`, crie um arquivo chamado `pagina.html` com o seguinte conteúdo:

```
<h1>Olá, mundo!</h1>
<p>Esta é minha página feita em Django</p>
```

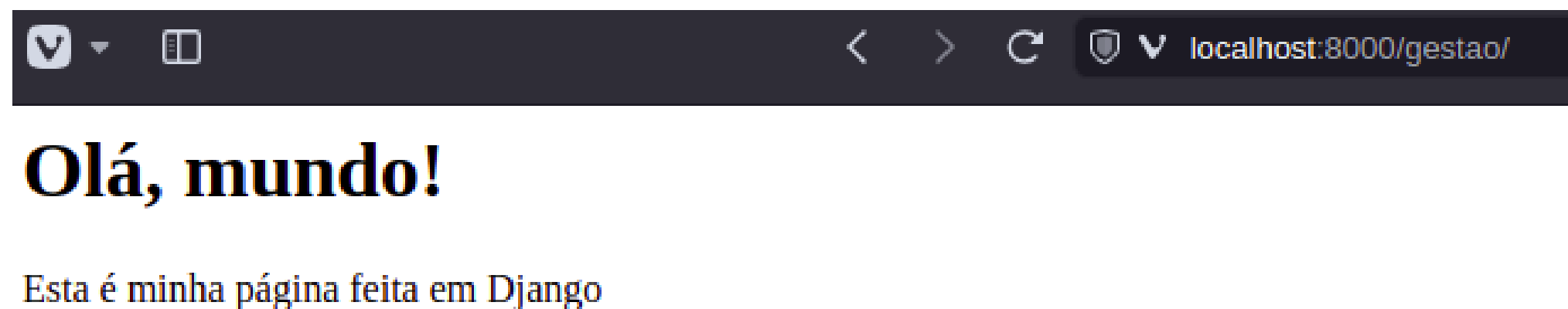
E altere seu arquivo `views.py` para ficar da seguinte forma:

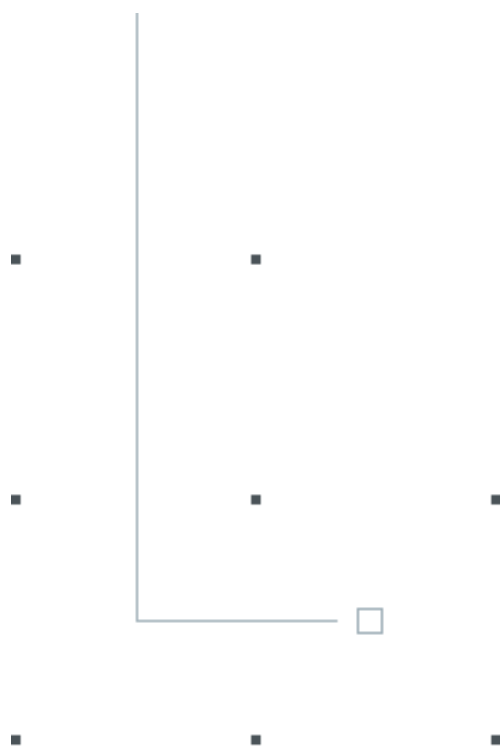
```
from django.shortcuts import render

# Create your views here
def index(request):
    return render(request, 'gestao/pagina.html')
```

# Templates

Agora ao acessar a url <http://localhost:8000/gestao> irá aparecer o seguinte conteúdo:





.

F

-

I

A

P

