1 Introdução

Neste material, vamos utilizar o arcabouço Django para desenvolver uma API REST para manipulação de filmes. Nossa API terá os seguintes endpoints.

GET /filmes (obtém a coleção de filmes)
GET /filmes/id (obtém os dados do filme cujo id foi especificado como parâmetro de path)
POST /filmes (armazena um novo filme enviado como JSON)
DELETE /filmes/id (apaga o filme cujo id foi especificado como parâmetro de path)

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

Nota. Se você já possuir um ambiente virtual com Django, pode fazer uso dele.

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.

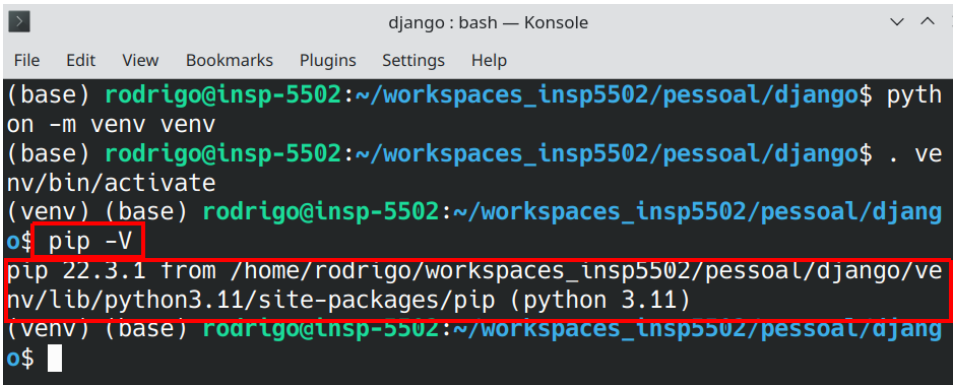
Depois da criação do ambiente virtual, é preciso ativá-lo. Usuários Windows, podem utilizar um terminal “cmd” ou um terminal “Powershell”. A tabela a seguir resume a forma como o ambiente virtual Python pode ser ativado em qualquer caso.

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

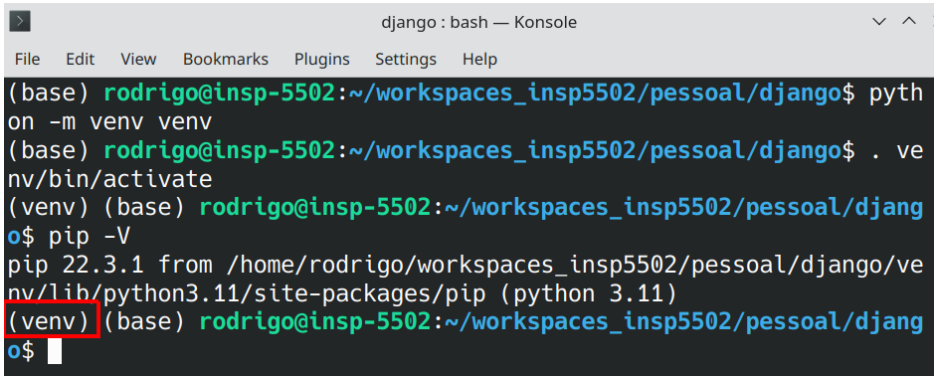
Em qualquer caso, você pode verificar se o ambiente foi ativado com sucesso com

pip -V

A pasta de seu ambiente virtual deve ser exibida.



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



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.

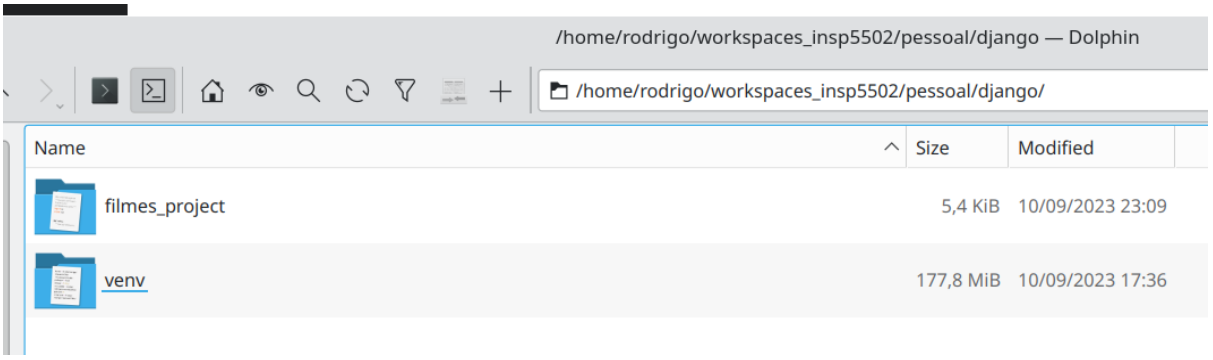
O próximo passo é criar um **projeto Django**. Um projeto Django é uma coleção de aplicações Django e configurações. Em geral, ele possui

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

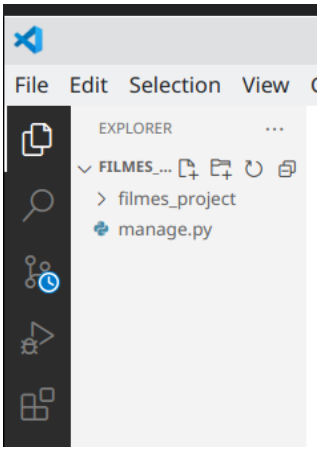
Podemos criar o projeto com

```
django-admin startproject filmes_project
```

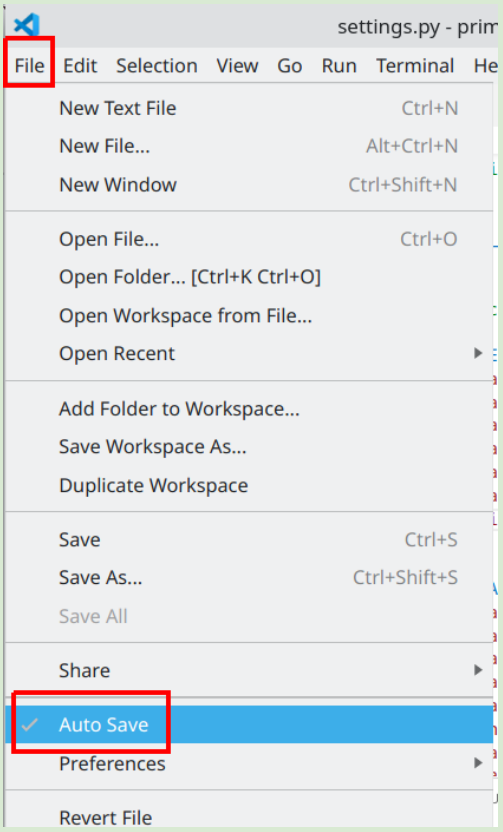
Uma pasta deve ter sido criada, ao lado da pasta que representa o seu ambiente virtual.



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.



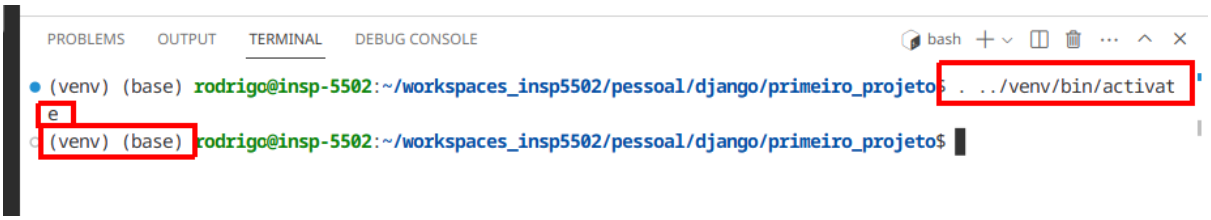
Nota. É recomendável manter o VS Code em modo de salvamento automático, clicando em File >> Auto Save.



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.



Use o comando apropriado para o seu sistema operacional.

2.2 (Novas dependências: **Django Rest Framework (DRF) e psycopg2 (driver PostgreSQL))** Para criar a API Rest, vamos utilizar o conhecido DRF. Se desejar saber mais sobre ele, visite

https://www.django-rest-framework.org/

Além disso, nossa aplicação se conectará a uma instância do PostgreSQL e, para tal, ela precisa ser capaz de se comunicar utilizando o protocolo do PostgreSQL. Ele é implementado por diferentes pacotes, que geralmente levam o nome de "driver". Neste material, vamos utilizar o pacote **psycopg2**. No terminal do VS Code, use

pip install django djangorestframework psycopg2

para instalar ambos.

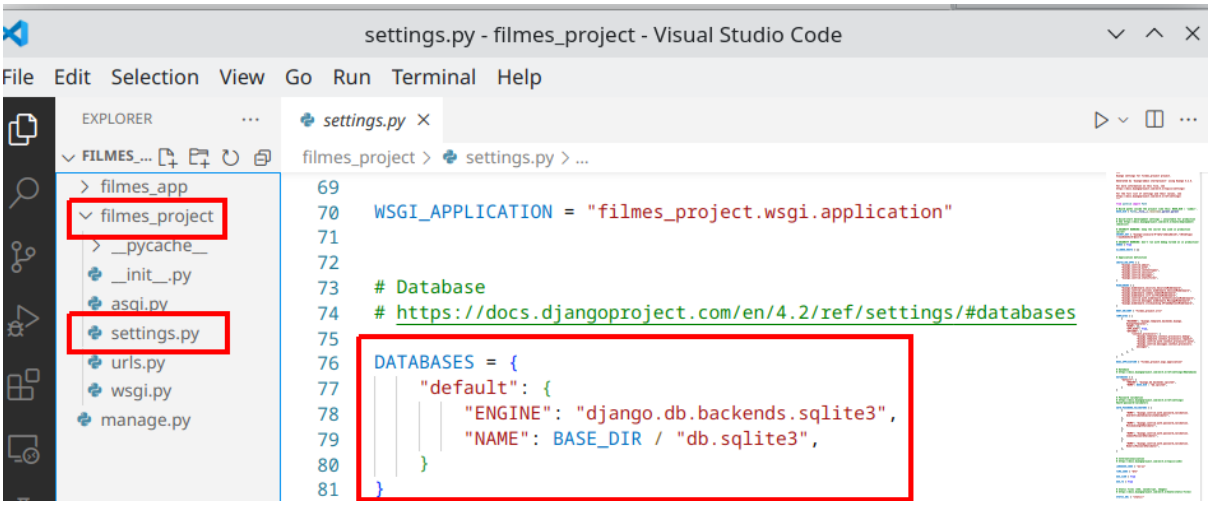
2.3 (Nova aplicação) Até então, temos apenas um projeto Django. Agora precisamos criar uma aplicação que fará parte dele. Para isso, use

python manage.py startapp filmes_app

2.4 (Adicionando aplicações ao projeto) No arquivo **settings.py do projeto**, precisamos adicionar a nossa aplicação como parte dele, na lista de INSTALLED_APPS. Como vamos usar o DRF, também precisamos adicionar uma aplicação chamada **rest_framework**. Veja.

```
...
ALLOWED_HOSTS = []

# Application definition

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

MIDDLEWARE = [
...
```

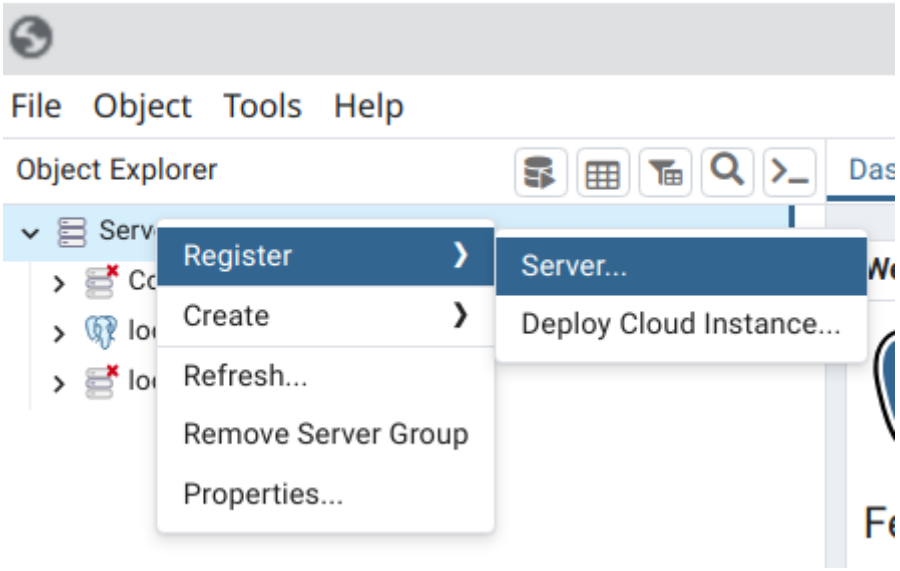
2.5 (Configurando os dados de acesso ao PostgreSQL) No arquivo **settings.py do projeto**, temos uma variável chamada **DATABASES**. Ela referencia um dicionário Python que contém configurações de acesso a bases de dados.



Observe que, por padrão, o Django usa o SQLite 3. Vamos alterar para utilizar o PostgreSQL. Veja como fica.

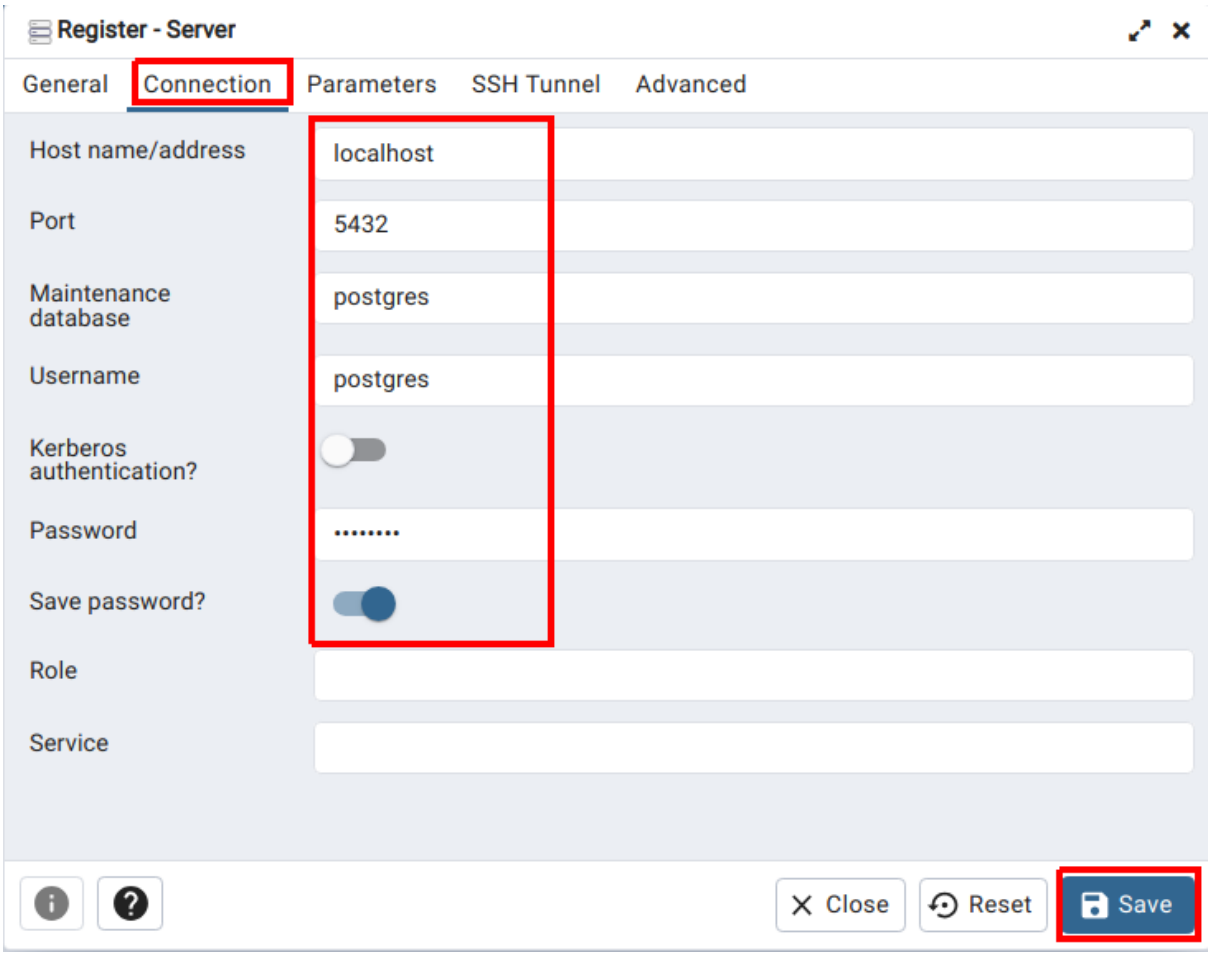
```
...
DATABASES = {
   'default': {
     'ENGINE': 'django.db.backends.postgresql',
     'NAME': 'pessoal_pdfs_rest_filmes',
     'USER': 'postgres',
     'PASSWORD': 'postgres',
     'HOST': 'localhost',
     'PORT': '5432',
   }
}
...
```

2.6 (Criando um database com o pgAdmin) Observe que estamos utilizando um nome específico para o database. Embora possa lidar com a criação de tabelas, o Django não cria o database. Devemos fazê-lo manualmente. Para isso, abra o pgAdmin. Clique com o direito em **Servers** e escolha **Register >> Server**.

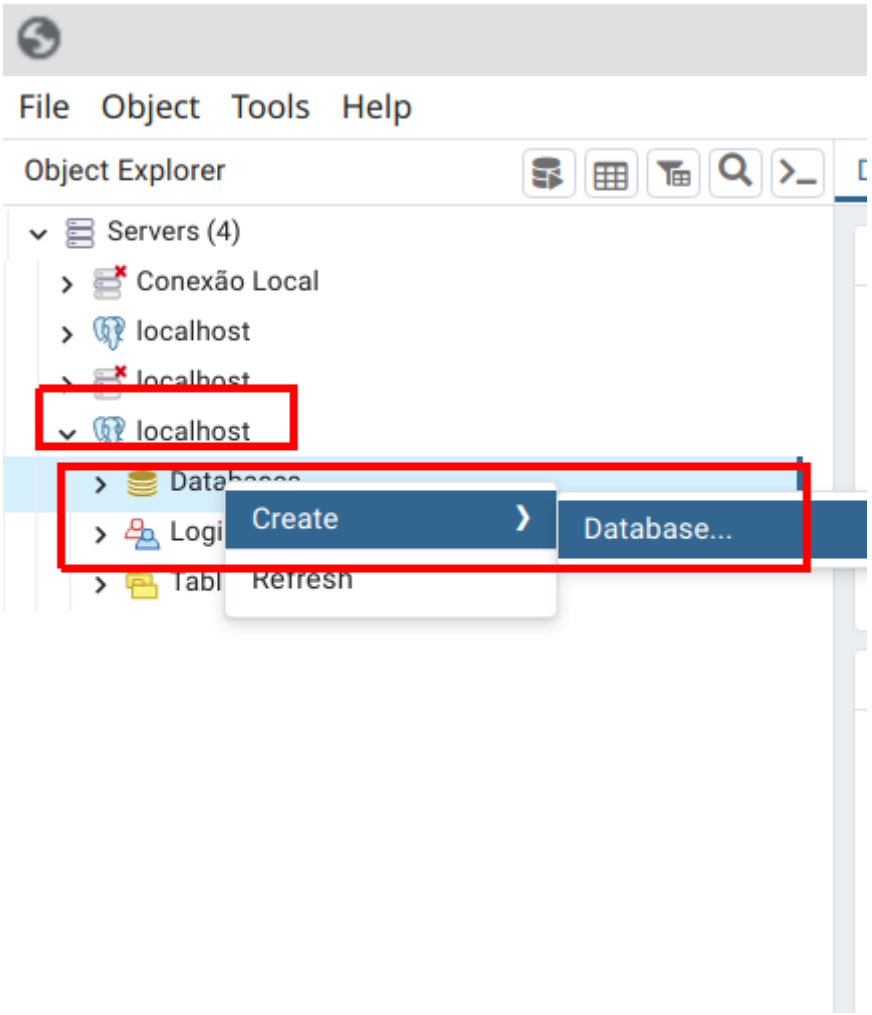


O nome pode ser algo como localhost.

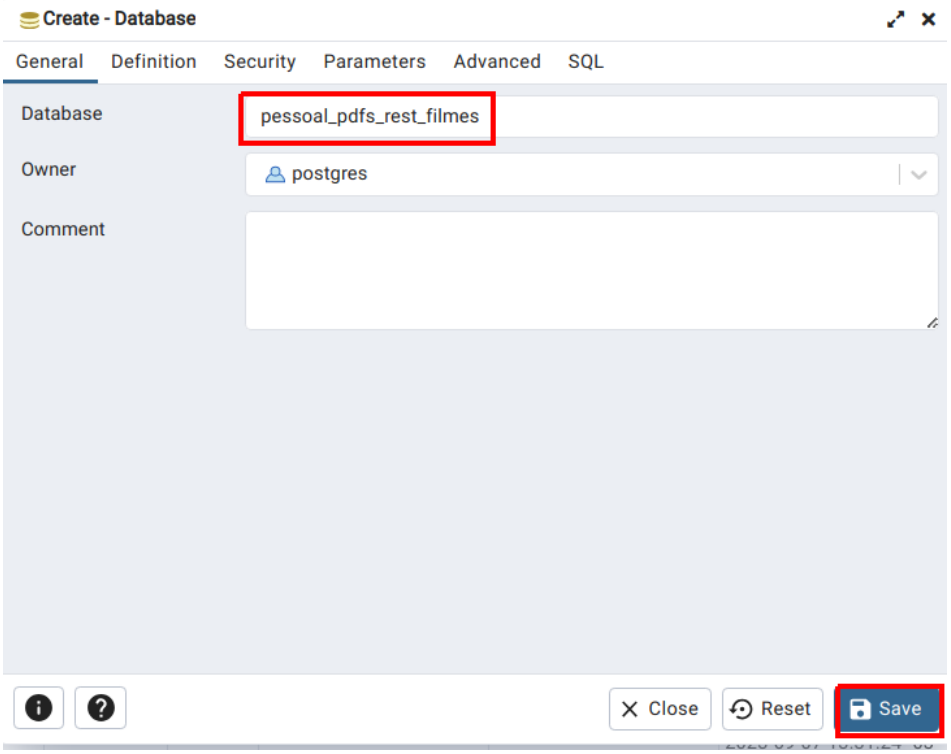
Na aba **Connection**, precisamos especificar os detalhes para acesso ao servidor local. Clique em **Save** a seguir.



Clique sobre **localhost** à direita, clique com o direito sobre **Databases** e escolha **Create >> Database.**



Coloque o mesmo nome que tenha especificado no arquivo **settings.py** do seu projeto Django. Clique em **Save** a seguir.



2.7 (Definindo uma classe de modelo para descrever o que é um filme) No arquivo **models.py da aplicação**, faça a definição de uma classe. Ela herda de models.Model e descreve as propriedades de interesse de um filme, além de aplicar validações eventualmente desejadas. Podemos também definir o método __str__, responsável por produzir uma representação textual dos objetos da classe. Observe que, neste momento, estamos fazendo uso do mecanismo de **mapeamento objeto relacional** provido pelo Django.

```
from django.db import models

# Create your models here.

class Filme(models.Model):
  titulo = models.CharField(max_length=100)
  descricao = models.TextField()
  diretor = models.CharField(max_length=100)

  def __str__(self):
    return self.titulo
```

2.8 (Explicando como converter um objeto Python em JSON e vice-versa: serializers) O DRF nos permite especificar a forma como um objeto Python deve ser convertido para JSON e vice-versa. Em geral, escolhemos os campos desejados e o trabalho "duro" fica todo por conta de classes utilitárias. Uma classe que desempenha tal papel é chamada de **Serializer**. Crie um arquivo chamado **serializers.py na pasta da aplicação**. Veja a definição de um serializer que converte objetos do tipo filme incluindo todos os seus campos.

```
from rest_framework import serializers
from .models import Filme
class FilmeSerializer (serializers.ModelSerializer):
  class Meta:
    model = Filme
    fields = '__all__'
```

2.9 (Views para lidar com as requisições) Nossa API possui endpoints que são acessados utilizando métodos comuns do protocolo HTTP, como GET, POST etc. O DRF oferece classes genéricas que implementam os detalhes de mais baixo nível do protocolo e simplificam bastante a nossa tarefa. Neste exemplo, vamos usar duas classes:

- **ListCreateAPIView**: para implementar os endpoints

GET /filmes
POST /filmes

Observe que o padrão de acesso é o mesmo, o que muda é o método do protocolo HTTP. Por isso faz sentido que uma única classe seja capaz de lidar com os dois endpoints.

- **RetrieveDestroyAPIView**: para implementar os endpoints

GET /filmes/{id}
DELETE /filmes/{id}

Novamente, o padrão de acesso é o mesmo, o que muda é o método do protocolo HTTP. Essa classe é capaz de lidar com ambos, como o nome sugere.

Nota. Há também os conhecidos **mixins**, que implementam métodos separadamente, sem fazer combinações com as classes que escolhemos. Veja mais em

https://www.django-rest-framework.org/api-guide/generic-views/

Observe que, em nossa implementação, fazemos uso de dois atributos. **Ambos são definidos pela classe da qual herdamos**.

- queryset: associamos a esse atributo a coleção sobre a qual desejamos operar. Tanto para indicar qual lista de objetos deve ser devolvida quanto para indicar à qual lista um novo objeto deve ser adicionado.

- **serializer_class**: associamos a esse atributo o nome de uma classe que é responsável por fazer eventuais validações e conversões de objeto Python para JSON e vice-versa. Ou seja, a classe Serializer que criamos há pouco.

Veja como fica. Use o arquivo **views.py da aplicação**.

```
from rest_framework import generics
from .models import Filme
from .serializers import FilmeSerializer

class FilmeListCreate(generics.ListCreateAPIView):
  queryset = Filme.objects.all()
  serializer_class = FilmeSerializer

class FilmeRetrieveDestroy(generics.RetrieveDestroyAPIView):
  queryset = Filme.objects.all()
  serializer_class = FilmeSerializer
```

2.10 (Mapeamento de URLs). Precisamos fazer o mapeamento URL/View como a seguir.

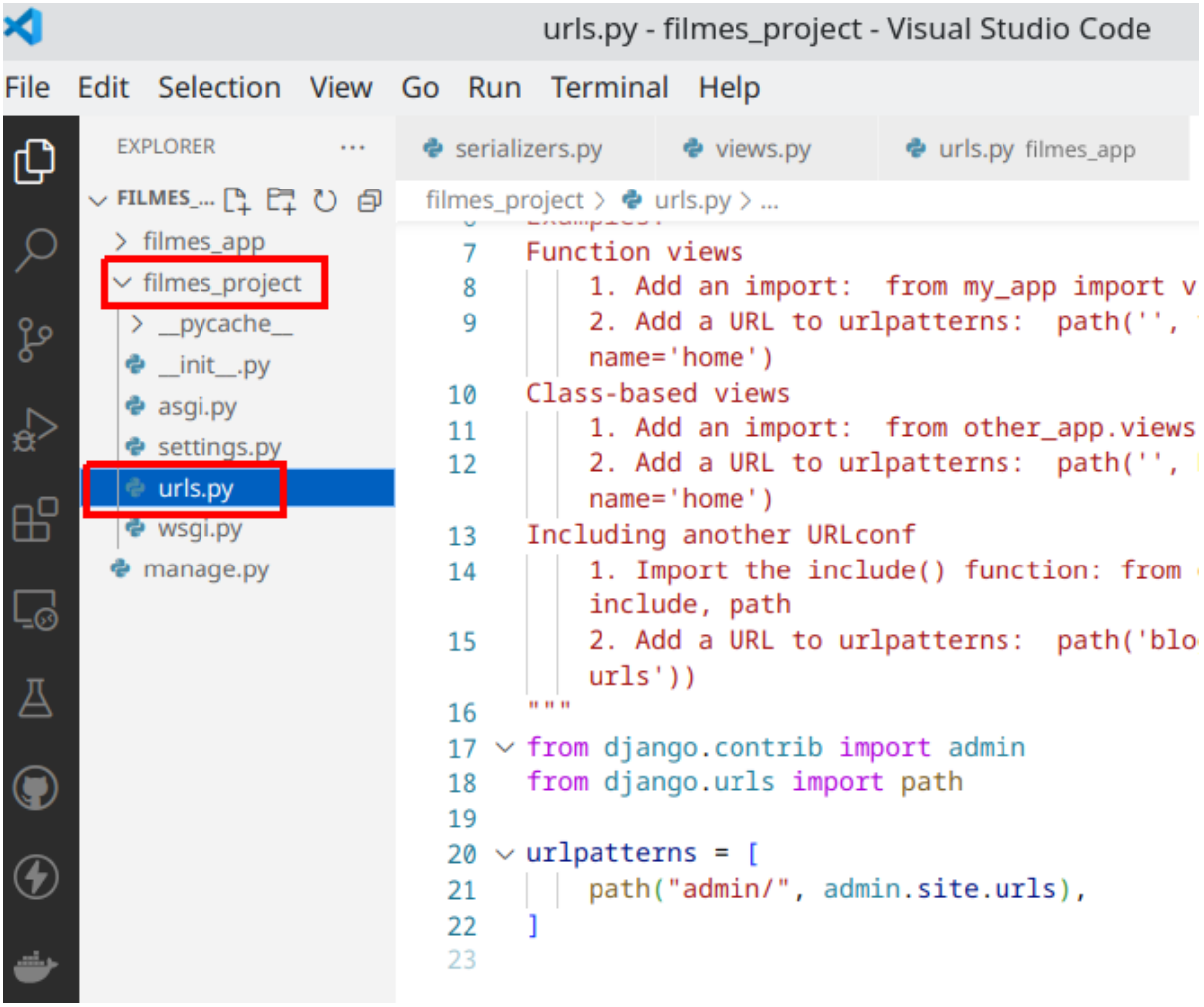
URL	View
filmes/	FilmeListCreate
filmes/<int:pk>	FilmeRetrieveDestroy

Nota. Usamos a notação <int:pk> para definir uma variável de path. Isso quer dizer que, no código, teremos acesso a uma variável chamada pk que dá acesso ao valor que tenha sido especificado na região em que ela foi definida na path. Além disso, há a validação de que ela deve ser int. O nome pk não é obrigatório, mas é bastante comum, especialmente pelo fato de, internamente, ele ser usado como parâmetro nomeado por métodos do DRF.

Crie um arquivo chamado **urls.py na pasta da aplicação**. Veja seu conteúdo.

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

urlpatterns = [
  path ('filmes/', views.FilmeListCreate.as_view(), name="filme-list-create"),
  path('filmes/<int:pk>', views.FilmeRetrieveDestroy.as_view(), name='filme-retrieve-destroy')
]
```

Lembre-se que é necessário incluir o módulo urls da aplicação que acabamos de criar no módulo principal do projeto. O arquivo também se chama **urls.py e se encontra na pasta do projeto**.



Veja como fica.

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

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

2.11 (Criando a estrutura da base em função da classe de Modelo e migrações) O mecanismo de mapeamento objeto relacional do Django permite que criemos as tabelas do banco de dados em função de classes de modelo que tenhamos definido. Além disso, contamos com um sistema de migração que permite alternarmos entre diferentes versões da base conforme a necessidade. Todo o histórico de alterações é mantido em tabelas também criadas na nossa base pelo Django. Temos dois comandos

- makemigrations: Detecta eventuais diferenças entre suas classes de modelo e a estrutura da base de dados atual. Cria arquivos que, se executados, fazem as transformações necessárias. Ou seja, aqui a base não é alterada ainda, o que dá maior flexibilidade ao desenvolvedor. São gerados arquivos de migração que ficam armazenados na pasta **migrations da aplicação**.

- migrate: executa os arquivos criados pelo comando makemigrations, alterando a estrutura da base.

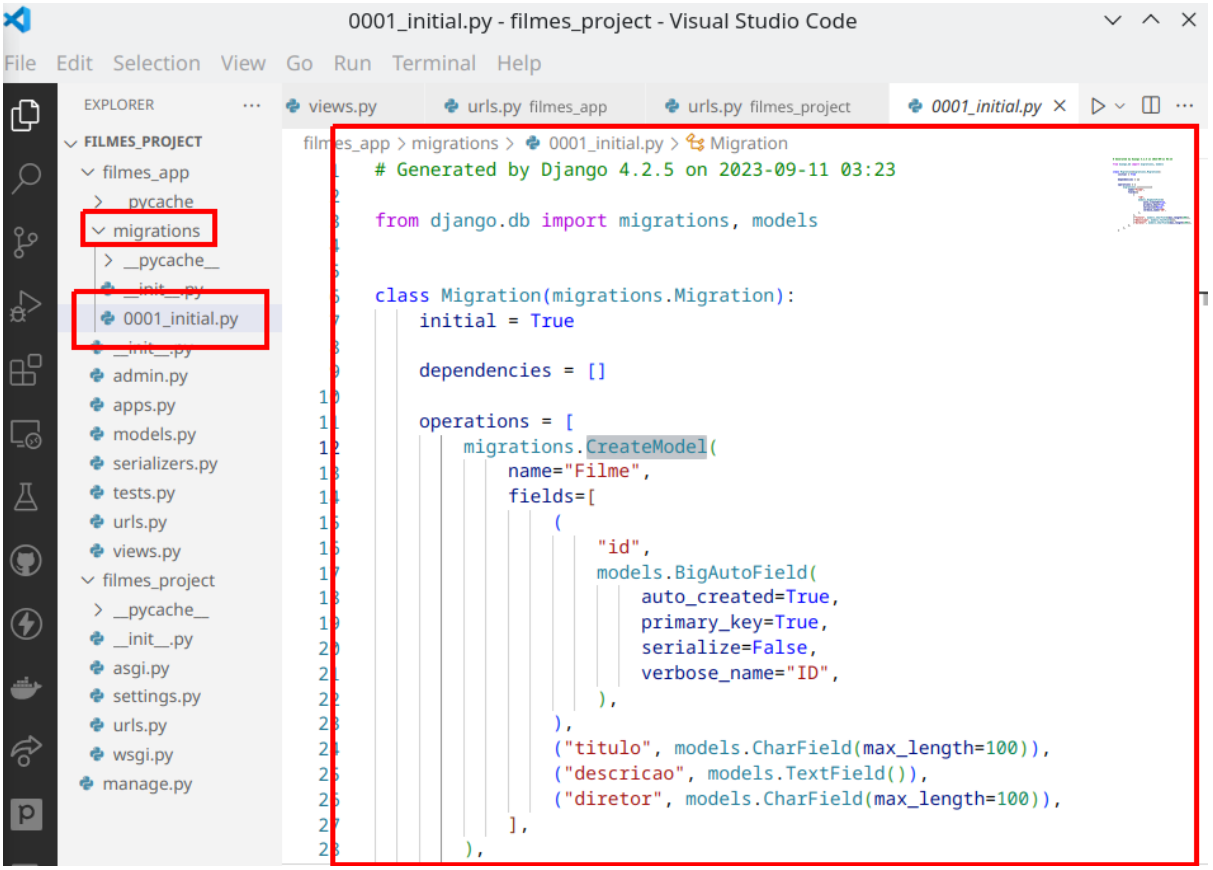
No terminal do VS Code, execute o primeiro com

python manage.py makemigrations filmes_app

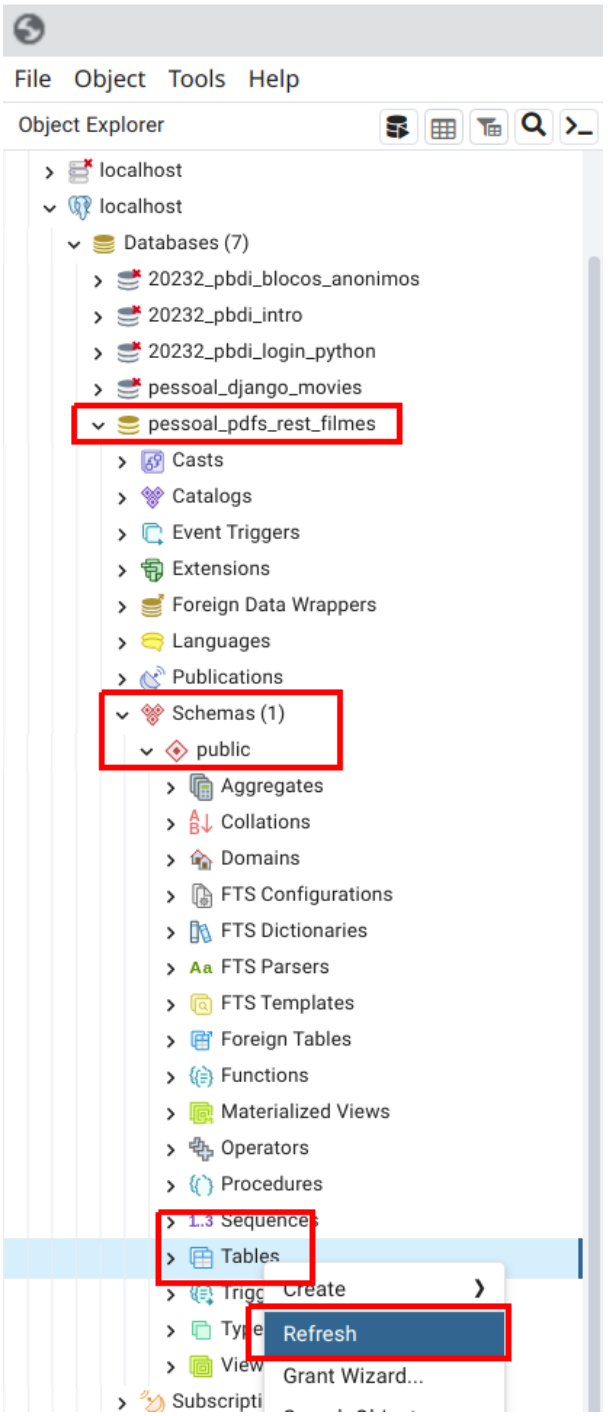
Veja o feedback textual dado no terminal.

```
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/filmes_project$ python manage.py makemi
grations filmes_app
Migrations for 'filmes_app':
  filmes_app/migrations/0001_initial.py
    - Create model Filme
```

Por curiosidade, expanda a pasta **migrations da aplicação.** Observe que um arquivo chamado **0001_initial.py** foi criado. Abra-o e veja seu conteúdo. Ali temos código Python que, quando executado, altera a estrutura da base. Não precisamos mexer neste arquivo, embora possamos.



Se você abrir o pgAdmin, expandir até encontrar a seção de tabelas e clicar em **Refresh**, como na figura a seguir, perceberá que nenhuma tabela foi criada ainda.



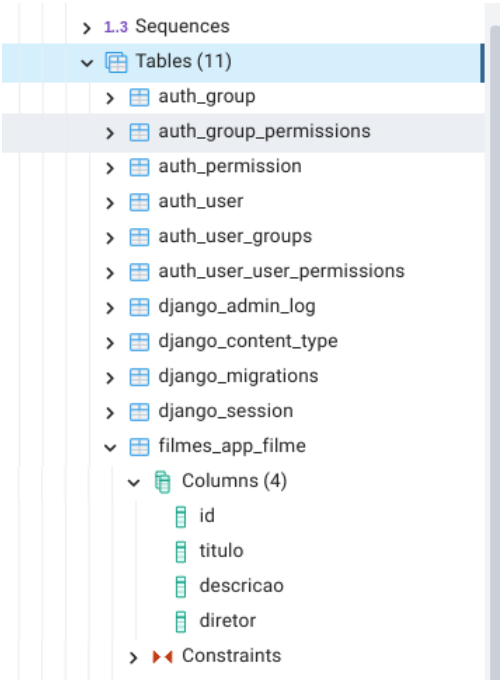
Execute o próximo comando no terminal do VS Code

python manage.py migrate

Veja o feedback textual no terminal.

```
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/filmes_project$ python manage.py migrat
e
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, filmes_app, 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 filmes_app.0001_initial... OK
  Applying sessions.0001_initial... OK
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/filmes_project$
```

Volte ao pgAdmin e clique **Refresh** sobre **Tables** novamente. Veja que foram criadas diversas tabelas. Neste momento, a maioria é referente ao próprio controle realizado pelo Django. Observe que uma delas serve para abrigar dados de filmes.



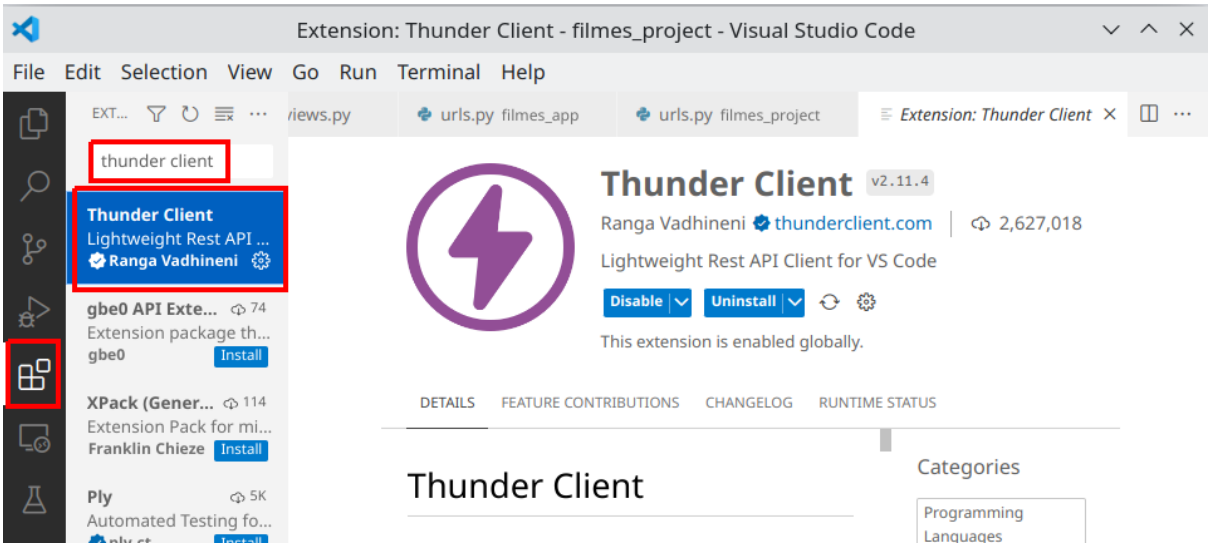
2.12 (Executando o servidor e disparando requisições com a Thunder Client) Agora podemos verificar se a aplicação está funcionando corretamente. Comece colocando o servidor em execução com

python manage.py runserver

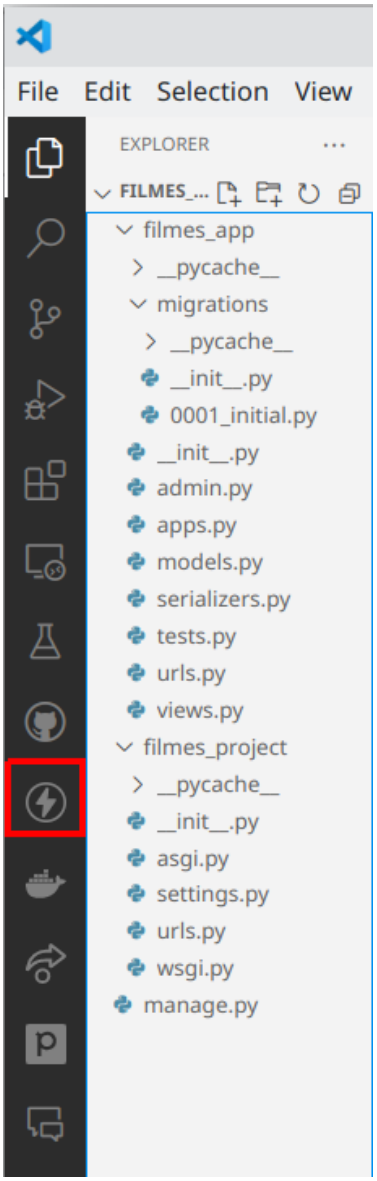
Veja o feedback textual esperado.

```
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/filmes_project$ python manage.py runser
ver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
September 11, 2023 - 03:35:11
Django version 4.2.5, using settings 'filmes_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

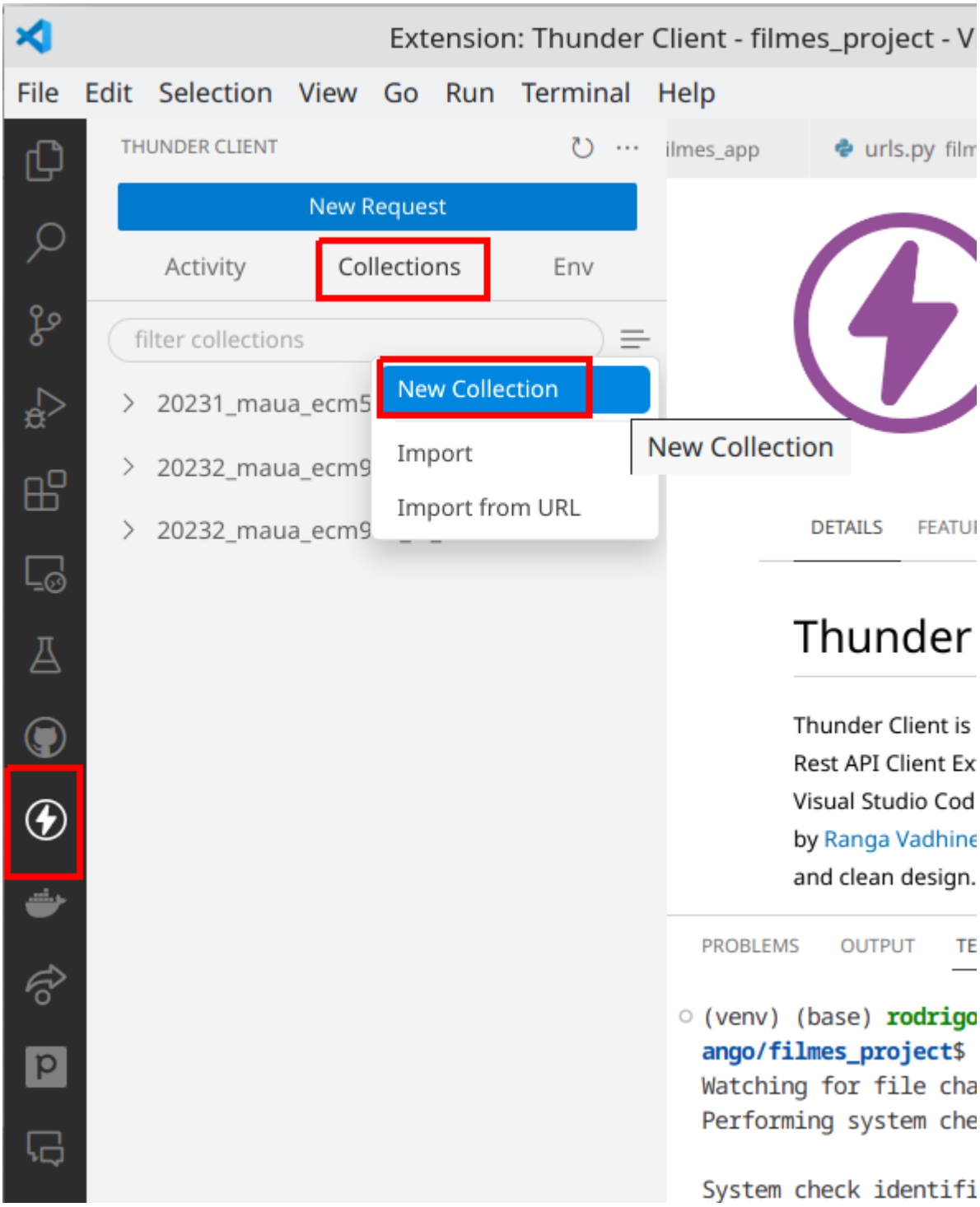
Há uma extensão para o VS Code chamada **Thunder Client**. Trata-se de um cliente HTTP que nos permite construir requisições mais detalhadamente do que faríamos com um navegador comum. É uma ferramenta essencial para o desenvolvedor. Se você ainda não tiver, faça a sua instalação.



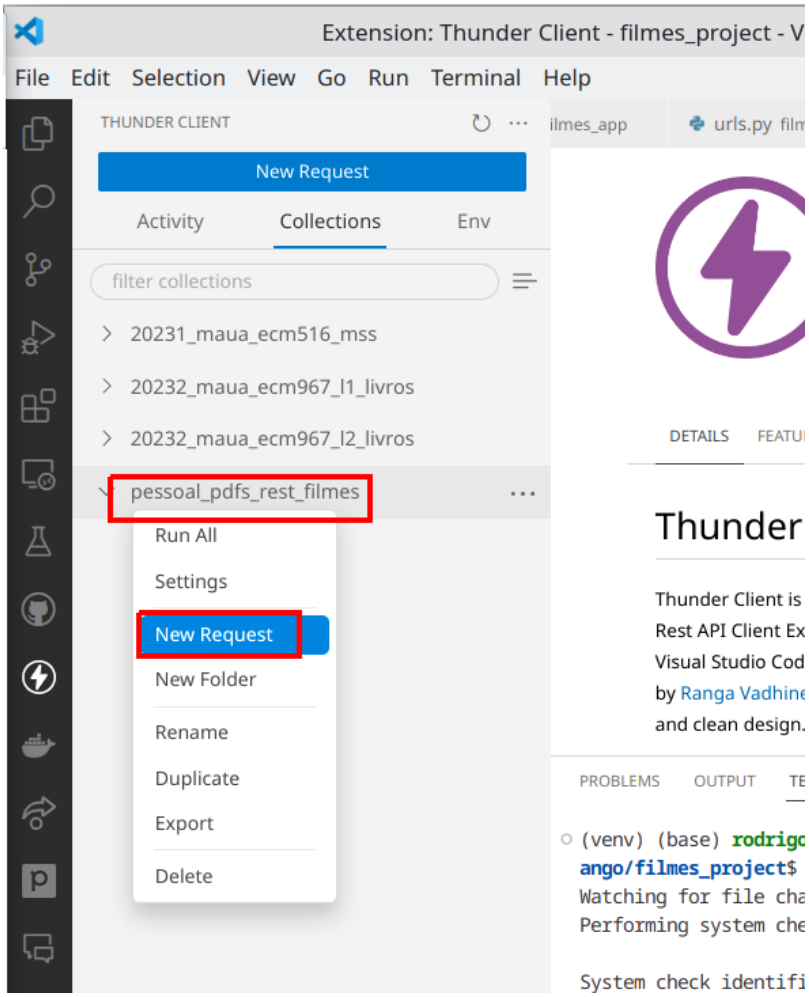
Uma vez que ela tenha sido instalada, você deve ser capaz de ver seu ícone na barra lateral esquerda do VS Code.



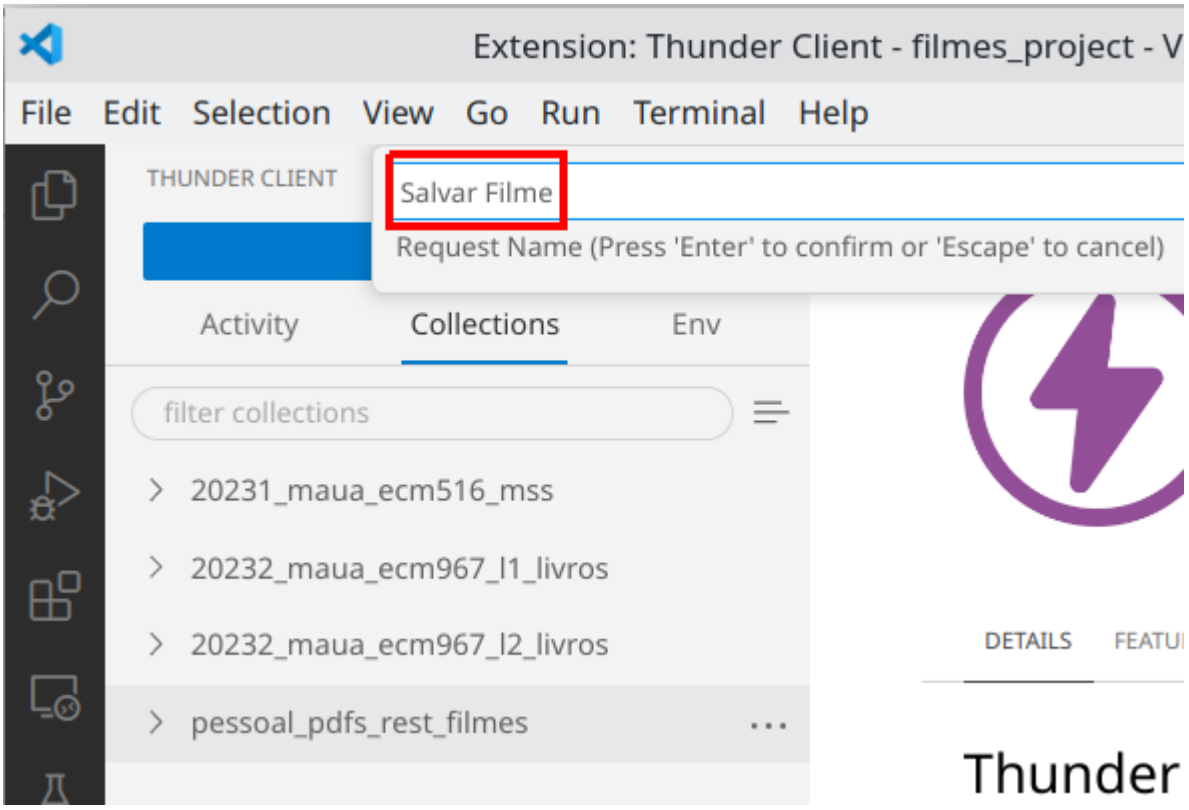
Comece criando uma nova coleção. Vamos utilizá-la para agrupar as requisições referentes a este projeto.



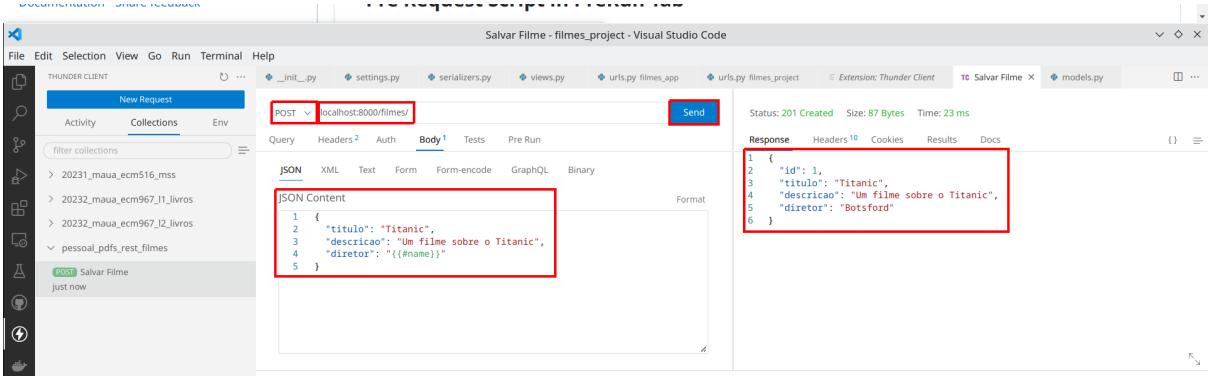
Depois de escolher um nome para a coleção, clique com o direito sobre seu nome e escolha **New Request**.



Dê um nome para a requisição e aperte Enter. Esta primeira requisição servirá para salvarmos um filme na base.



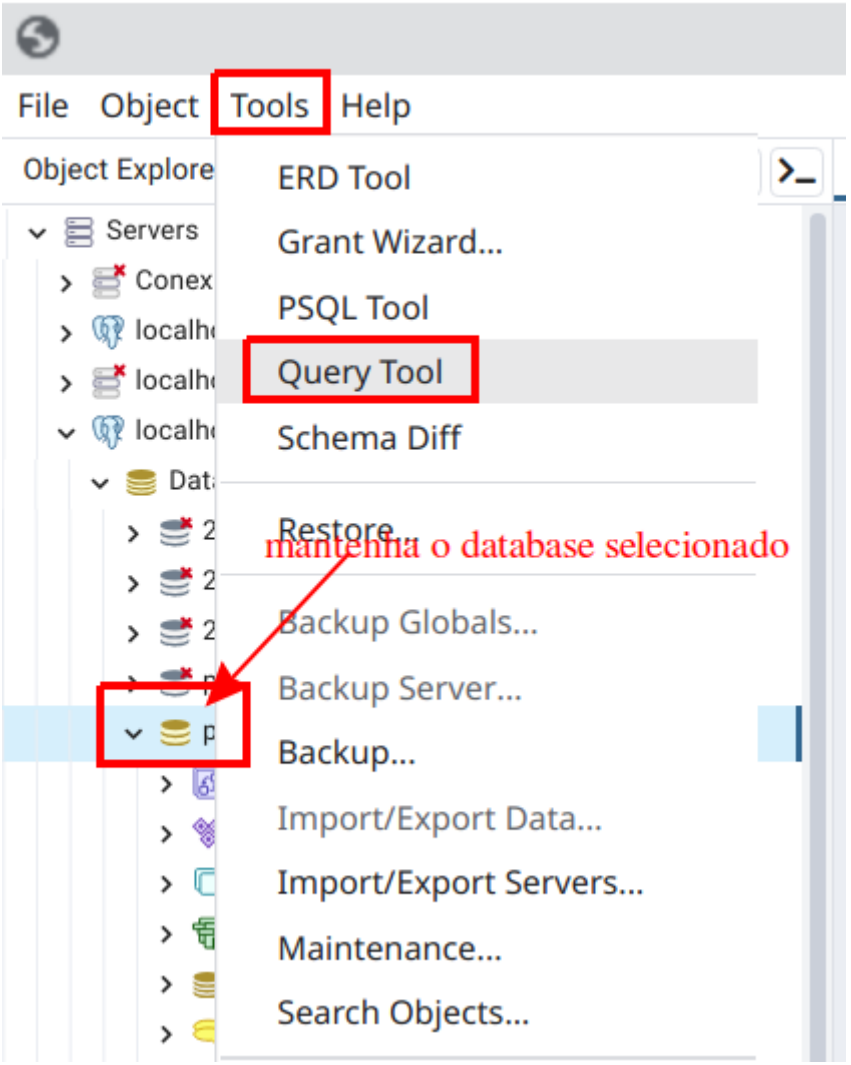
Faça os ajustes destacados a seguir e clique em **Send** para fazer uma requisição.



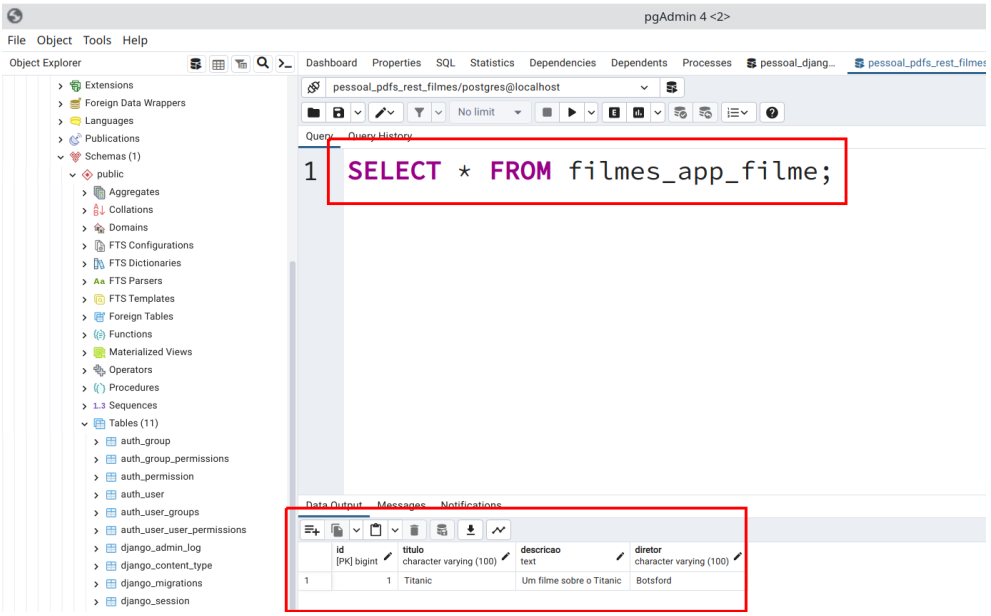
Nota. Estamos utilizando a notação {{#name}} da Thunder Client para gerar um nome aleatório. Ela possui outras construções semelhantes, que permitem gerar números, datas etc. Veja mais em

https://github.com/rangav/thunder-client-support#system-variables

No pgAdmin, **mantenha seu database selecionado (basta clicar no seu nome)** e clique em **Tools >> Query Tool**.



Execute um SELECT e verifique que os dados foram armazenados na tabela.



Como exercício, crie as demais requisições na Thunder Client e faça novos testes.

2.13 (Senhas e valores hard-coded ou "chumbadas" no código e no controle de versão são indesejadas)

Considere os seguintes valores de que projetos geralmente dependem para funcionar

- senhas
- chaves de API
- URL de acesso à base de dados

entre muitos outros.

Estes valores não devem ser armazenados diretamente no código Python por, pelo menos, duas razões:

- são valores que **variam em função do ambiente**: se estamos em tempo de desenvolvimento, provavelmente estamos acessando uma base própria para isso, apenas para os testes do desenvolvedor, isolada de outros ambientes, especialmente de produção

- são valores que **não devem fazer parte do controle de versão**, especialmente se estivermos utilizando um repositório público.

As diferentes linguagens de programação possuem diferentes mecanismos para lidar com isso. Em Python, algumas opções são:

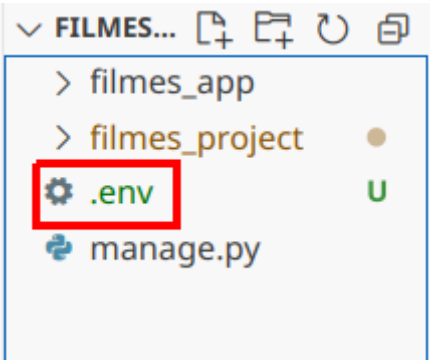
- **python-decouple**: https://pypi.org/project/python-decouple/
- **python-dotenv**: https://pypi.org/project/python-dotenv/
- **django-environ**: https://pypi.org/project/django-environ/

Entre outros.

Neste material, utilizaremos o pacote **django-environ**. O primeiro passo é fazer a sua instalação.

```
pip install django-environ
```

A seguir, criamos um arquivo chamado **.env** na raiz do projeto, lado a lado com o arquivo **manage.py.**



Quando um projeto Django é colocado em execução, o arquivo **settings.py** é executado automaticamente. Por isso, vamos fazer a leitura do arquivo .env nele. Para tal

- importamos o módulo **environ**
- construímos um objeto do tipo environ.Env
- usamos o método read_env para ler o conteúdo

Observe que vamos utilizar o "diretório base" da aplicação, que é a pasta que contém o arquivo .env que criamos. Ele já vem definido por padrão e se encontra na variável BASE_DIR.

> **Nota**. Há uma sobrecarga do operador / para objetos do tipo Path do módulo pathlib. Quando o operador / opera com operandos do tipo Path, ele significa "separador de path independente de sistema operacional".

Veja como ficou até agora.

```
...
from pathlib import Path
import environ

env = environ.Env()

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
environ.Env.read_env(BASE_DIR / Path(".env"))
...
```

Ainda no arquivo **settings.py**, observe que temos dois valores.

SECRET_KEY: Utilizada pelo Django em funcionalidades como

- Assinatura criptográfica de dados mantidos em sessão, para que seja possível verificar que os dados enviados pelo cliente não foram adulterados.

- Geração de tokens para reconfiguração de senha perdida

DEBUG: valor booleano que indica que a aplicação está em modo de depuração. Devemos configurá-la como True apenas em tempo de desenvolvimento. Veja algumas de suas características.

- Quando vale True, o Django exibe a pilha de chamada de métodos quando um erro acontece.

- Mantém as queries de consulta ao banco em memória, o que é bom para depuração mas causa impacto no desempenho da aplicação, algo desnecessário em tempo de produção.

- Por questões de segurança, ela nunca deve valer True em tempo de produção, já que uma pessoa mal intencionada poderia utilizar de técnicas para tentar atacar a aplicação e encontrar as informações mantidas em memória sobre seu funcionamento.

Assim, vamos definir ambas no arquivo **.env**.

```
SECRET_KEY=django-insecure-9^t&7y^1hb*j8d=15+_^l0tnb7jqy)!-oyu8o$o5u)5-@1c*
-4
DEBUG=True
```

A seguir, passamos a utilizá-las no arquivo **settings.py**.

```
...

env = environ.Env()
# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
environ.Env.read_env(BASE_DIR / Path(".env"))

# Quick-start development settings - unsuitable for production
# See
https://docs.djangoproject.com/en/4.2/howto/deployment/checklist/
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = env('SECRET_KEY')
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = env('DEBUG')
...
```

Se desejar, você pode testar a configuração exibindo os valores das variáveis.

```
...
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = env('SECRET_KEY')
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = env('DEBUG')
print(SECRET_KEY, DEBUG)
...
```

Para executar e ver o resultado, use

python filmes_project/settings.py

Observe.

```
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/fil
mes_project$ python filmes_project/settings.py
django-insecure-9^t&7y^1hb*j8d=15+_^l0tnb7jqy)!-oyu8o$o5u)5-@1c*-4 True
(venv) (base) rodrigo@insp-5502:~/workspaces_insp5502/pessoal/django/fil
mes_project$
```

Depois do teste, você pode **remover a instrução print**.

Também podemos fazer a construção do objeto Env mantendo algumas variáveis com valor padrão. Elas terão aquele valor a menos que o arquivo .env indique outro. Como é recomendável usar DEBUG=True apenas em ambiente de desenvolvimento, podemos adotar a seguinte estratégia:

- Ao construir o objeto Env, indicamos que o valor de DEBUG é igual a false.

- Redefinimos o valor de DEBUG (como já está feito neste exemplo) para False no arquivo .env.

Fica assim. Observe que atribuímos à variável DEBUG uma tupla: o primeiro valor indica o tipo ao qual o segundo será convertido.

```
from pathlib import Path
import environ

env = environ.Env(
  DEBUG = (bool, False)
)
```

Neste exemplo, o valor que prevalece é aquele definido no arquivo .env. Se não houver definição no arquivo .env, fica valendo o valor definido na construção do objeto .env.

Nossa aplicação possui **dados de acesso a bases de dados**. Estes também são variáveis de ambiente, já que podem variar em função do ambiente (desenvolvimento, produção etc). Assim, é interessante fazer a sua definição num arquivo isolado, como o arquivo .env. Veja como fica o arquivo **.env**.

Nota. Um projeto Django pode manipular múltiplas bases de dados. Por isso, ao especificar as chaves no arquivo .env, vamos qualificá-las com a palavra "DEFAULT", indicando que aquelas configurações se referem à base de dados a ser utilizada por padrão. Caso desejemos especificar novas variáveis para outras bases de dados, utilizaremos nomes diferentes para qualificá-las.

```
SECRET_KEY=django-insecure-9^t&7y^1hb*j8d=15+_^l0tnb7jqy)!-oyu8o$o5u)
5-@1c*-4
DEBUG=TRUE
DATABASE_DEFAULT_NAME=pessoal_pdfs_rest_filmes
DATABASE_DEFAULT_USER=postgres
DATABASE_DEFAULT_PASSWORD=postgres
DATABASE_DEFAULT_HOST=localhost
DATABASE_DEFAULT_PORT=5432
```

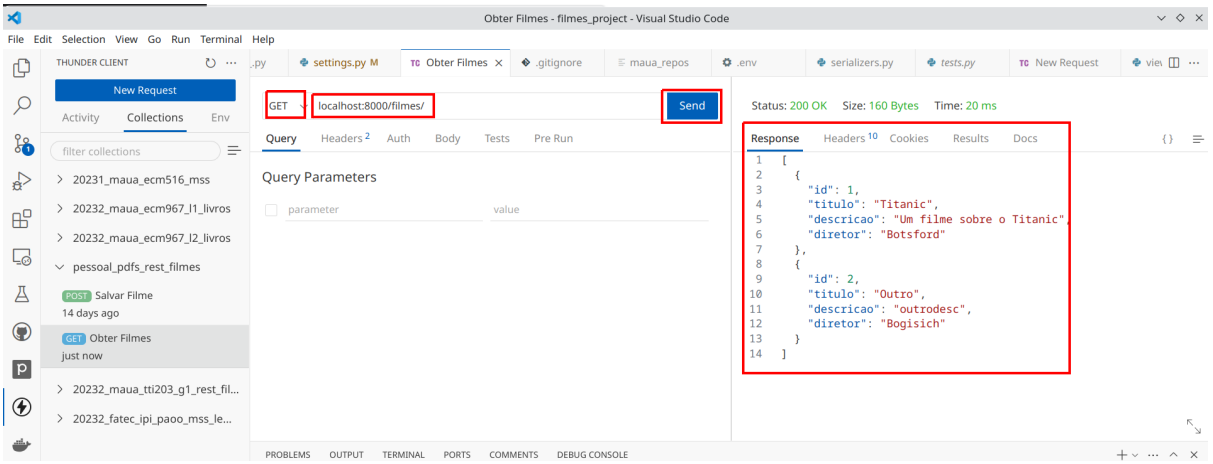
A seguir, no arquivo **settings.py**, passamos a utilizá-las.

```
...
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': env('DATABASE_DEFAULT_NAME'),
    'USER': env('DATABASE_DEFAULT_USER'),
    'PASSWORD': env('DATABASE_DEFAULT_PASSWORD'),
    'HOST': env('DATABASE_DEFAULT_HOST'),
    'PORT': env('DATABASE_DEFAULT_PORT'),
  }
}
...
```

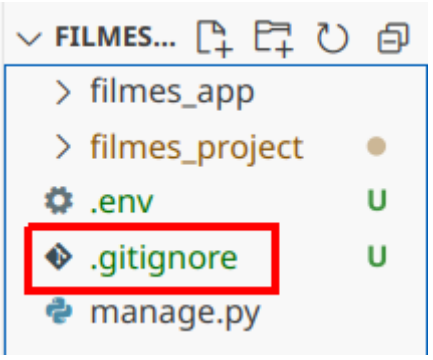
Execute a aplicação com

python manage.py runserver

e faça nova requisição com a Thunder Client, certificando-se de que os novos ajustes estão corretos.



É fundamental utilizarmos um arquivo chamado **.gitignore** na raiz da aplicação e, entre outras coisas, especificar que o arquivo .env não deve entrar no controle de versão. Caso ainda não possua um arquivo .gitignore, crie um na raiz.



Além do arquivo .env, há diversos outros arquivos que podemos estar interessados em especificar num arquivo .gitignore de um projeto Django. Veja um possível conteúdo para ele.

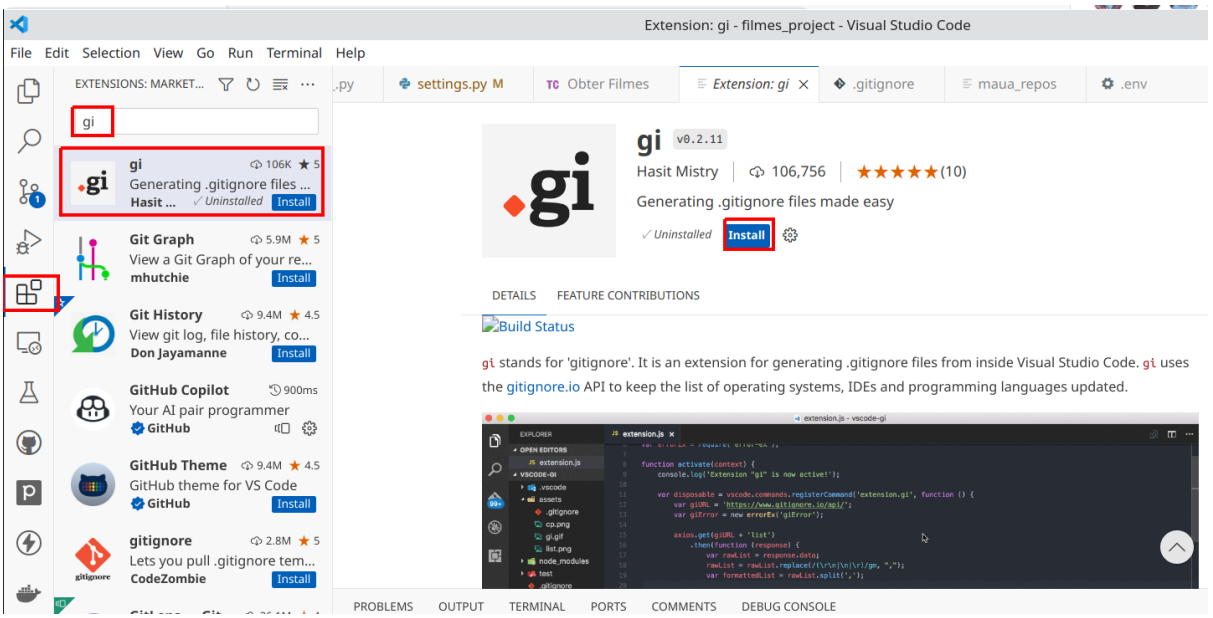
```
#pasta com código compilado (bytecode) em cache
__pycache__/
#código compilado ou arquivos para otimização
*.py[cod]
#código compilado para Jython
*$py.class

#a base de dados SQLite
db.sqlite3
```

```
#o arquivo .env
.env

#do MacOS
.DS_Store

#arquivos de log
*.log
```

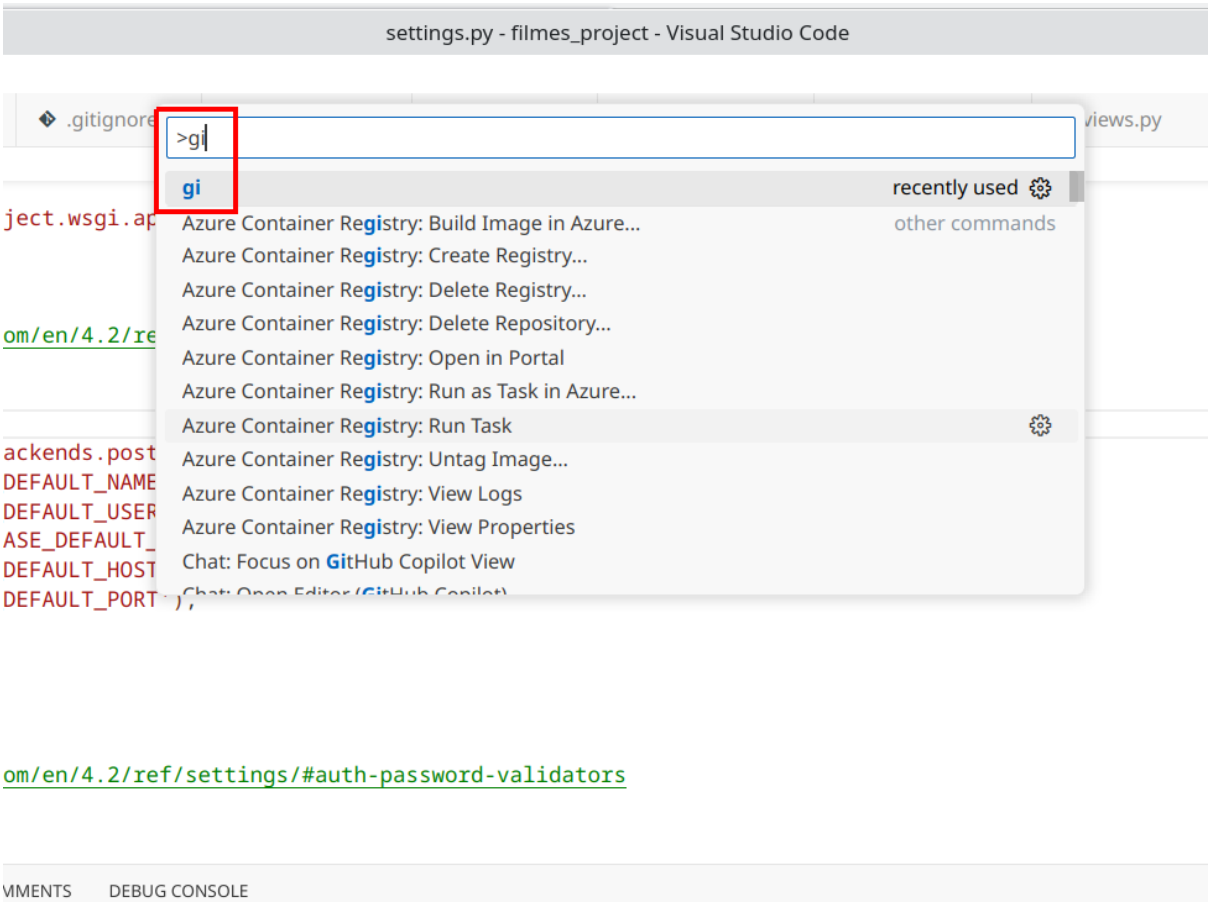
Há diversos outros arquivos que, ao longo do tempo, podem ser adicionados ao .gitignore, conforme o projeto passa a utilizar mais recursos. O site a seguir oferece uma boa ferramenta geradora de arquivos .gitignore.

https://www.toptal.com/developers/gitignore

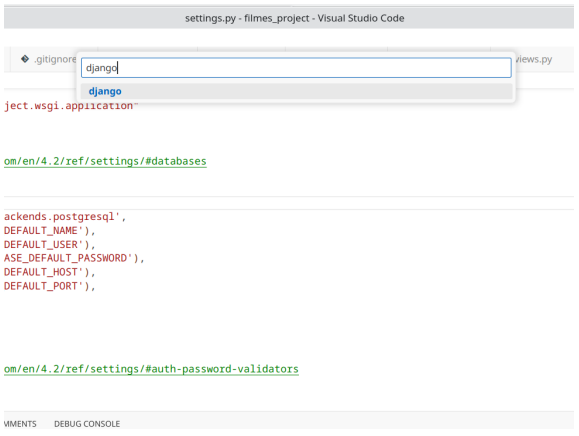
Basta buscar pelo tipo desejado. Caso deseje testar, basta buscar por Django. Há também uma extensão para o VS Code, caso deseje verificar. Seu nome é “gi”.



Para utilizá-la, basta apertar CTRL+SHIFT+P e buscar por gi. Talvez você precise “rolar” a lista até encontrar a extensão de nome gi.



Depois de selecioná-la, busque pelo tipo desejado. Neste caso, Django. Você pode buscar por nome de IDE (VS Code, Eclipse etc), nome de linguagem de programação (Java, C++ etc) e mais.



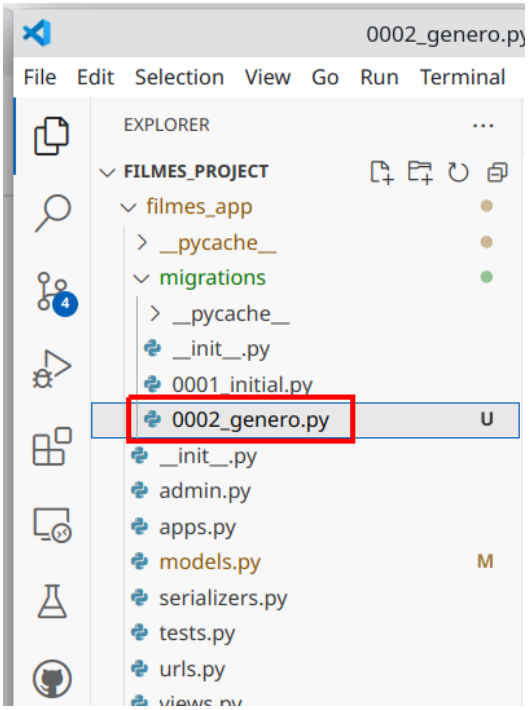
2.14 (Relacionamento 1xN: Um filme tem um gênero. Um gênero pode estar associado a N filmes) Neste exemplo, vamos adicionar um novo modelo ao projeto Django. Passaremos a lidar com objetos do tipo Gênero, os quais terão id e descrição. Além disso, há um relacionamento 1xN entre gênero e filme. Abra o arquivo **models.py** da aplicação e defina a nova classe de modelo, responsável por dizer o que é um Gênero. Observe que, para testes futuros, já sobrescrevemos o método __str__.

```
from django.db import models

# Create your models here.

  class Genero(models.Model):
    descricao = models.CharField(max_length=100)

    def __str__(self):
      return self.descricao


class Filme(models.Model):
  titulo = models.CharField(max_length=100)
  descricao = models.TextField()
  diretor = models.CharField(max_length=100)

  def __str__(self):
    return self.titulo
```

A seguir, vamos criar um arquivo de migração para que a tabela correspondente à classe Genero seja criada.

python manage.py makemigrations

Observe que um novo arquivo de migração foi criado.



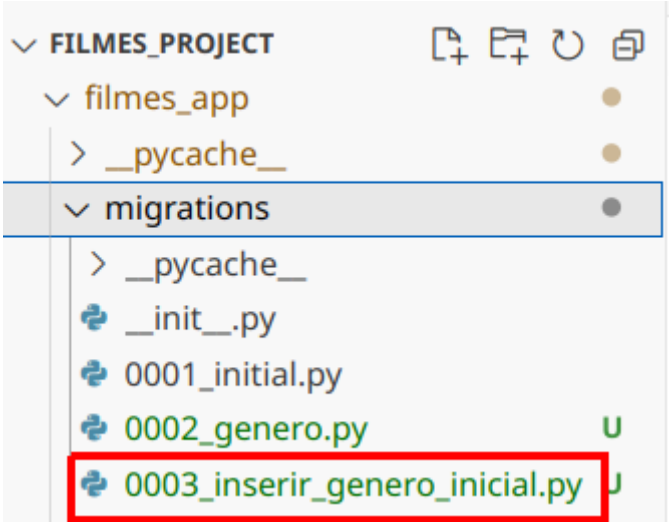
Por curiosidade, abra o arquivo e inspecione o seu conteúdo. Apenas observe, não há nada para alterar.

```
3   from django.db import migrations, models
4
5
6   class Migration(migrations.Migration):
7       dependencies = [
8           ("filmes_app", "0001_initial"),
9       ]
10
11      operations = [
12          migrations.CreateModel(
13              name="Genero",
14              fields=[
15                  (
16                      "id",
17                      models.BigAutoField(
18                          auto_created=True,
19                          primary_key=True,
20                          serialize=False,
21                          verbose_name="ID",
22                      ),
23                  ),
24                  ("descricao", models.CharField(max_length=100)),
25              ],
26          ),
27      ]
```

Antes de executar a **migração, vamos criar uma nova, inicialmente vazia**. Ela terá como finalidade **fazer o cadastro de um novo gênero**. Para isso, use

python manage.py makemigrations filmes_app --empty --name inserir_genero_inicial

Veja o arquivo criado.



Como esta é uma migração vazia, vamos abrir o arquivo e escrever código explicando o que ela deve fazer, quando aplicada. Começamos definindo uma função que

- obtém uma referência o modelo Gênero
- utiliza o método create de sua propriedade objects a fim de criar um novo gênero

```
from django.db import migrations

  def inserir_genero_inicial(apps, schema_editor):
    Genero = apps.get_model('filmes_app', 'Genero')
    Genero.objects.create(descricao="Romance")

  class Migration(migrations.Migration):
    dependencies = [
      ("filmes_app", "0002_genero"),
    ]

operations = []
```

Observe que a migração possui um campo chamado dependencies. Ele indica que ela somente pode ser executada depois de a migração ali especificada ter sido executada. Isso faz sentido, afinal, um gênero somente pode ser criado caso a tabela capaz de armazená-lo exista.

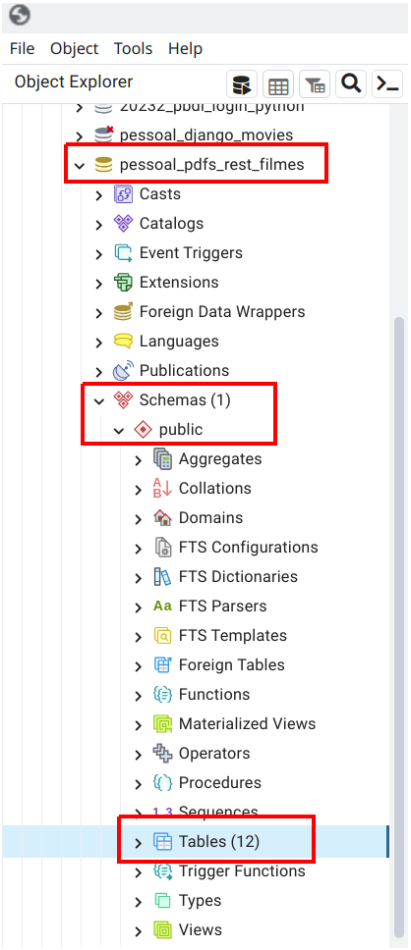
A seguir, na lista **operations**, incluímos a função que criamos anteriormente, responsável pelo cadastro do novo gênero.

```
from django.db import migrations

def inserir_genero_inicial(apps, schema_editor):
  Genero = apps.get_model('filmes_app', 'Genero')
  Genero.objects.create(descricao="Romance")

class Migration(migrations.Migration):
  dependencies = [
    ("filmes_app", "0002_genero"),
  ]

  operations = [
    migrations.RunPython(inserir_genero_inicial)
  ]
```

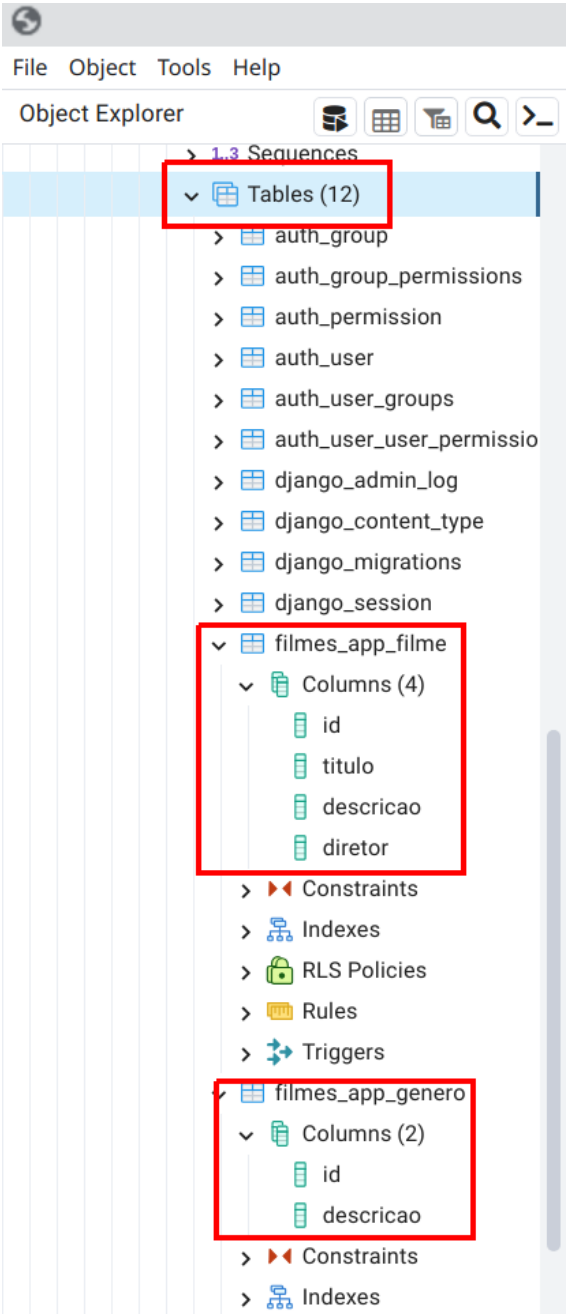
Depois disso, podemos executar as migrações.

python manage.py migrate

Já no pgAdmin, encontre o seu database, clique sobre ele, expanda Schemas >> public e encontre Tables.



Expanda Tables, encontre as tabelas referentes a filmes e a generos e veja as suas colunas.


File Object Tools Help
Object Explorer
1.3 Sequences
Tables (12)
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissio
django_admin_log
django_content_type
django_migrations
django_session
filmes_app_filme
Columns (4)
id
titulo
descricao
diretor
Constraints
Indexes
RLS Policies
Rules
Triggers
filmes_app_genero
Columns (2)
id
descricao
Constraints
Indexes

A seguir, precisamos estabelecer o relacionamento entre filme e gênero. Para isso, vamos usar uma chave estrangeira. Cada filme tem um novo campo. Ele referencia o id do gênero a que aquele filme está associado.

Nota. Estamos utilizando o valor models.CASCADE associado à propriedade on_delete. Isso quer dizer que, caso um gênero seja apagado, todos os filmes que estiverem associados a ele também serão apagados. Há outros possíveis valores como models.SET_NULL (para deixar cada filme sem gênero) e models.PROTECT(para não deixar que uma remoção de gênero aconteça, caso exista pelo menos um filme associado a ele.

Nota. Caso desejássemos que filmes pudessem existir sem gênero (relacionamento opcional, poderíamos usar null=True. Observe.

```
genero = models.ForeignKey(Genero, on_delete=models.CASCADE, null=True)
```

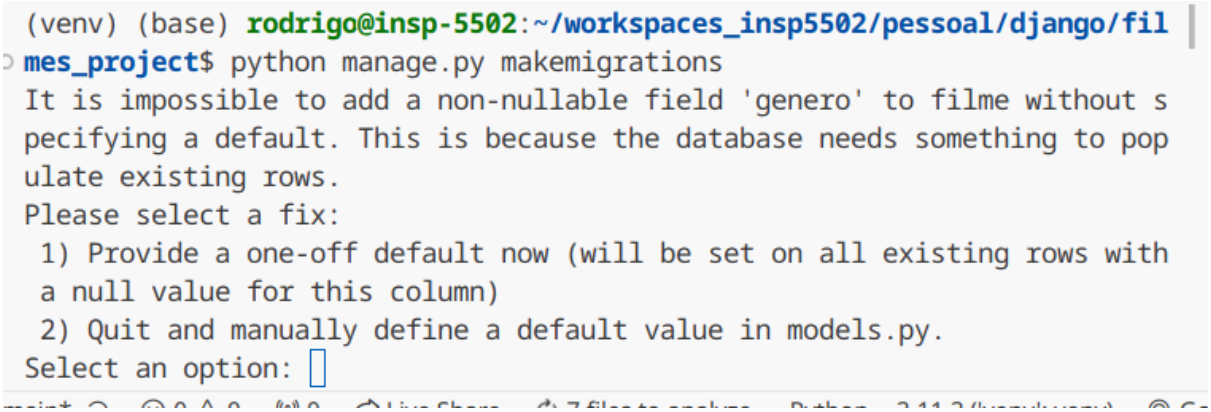
No momento, entretanto, vamos manter o padrão (que é False) e verificar o que acontece quando tentarmos executar uma nova migração.

```
from django.db import models

# Create your models here.
class Genero(models.Model):
descricao: models.CharField(max_length=100)
def __str__(self):
return self.descricao

class Filme(models.Model):
titulo = models.CharField(max_length=100)
descricao = models.TextField()
diretor = models.CharField(max_length=100)
genero = models.ForeignKey(Genero, on_delete=models.CASCADE)

def __str__(self):
return self.titulo
```

Como fizemos alterações nas classes de modelo, precisamos realizar o processo de migração novamente. Primeiro, geramos o código Python que fará as alterações necessárias na base, sem ainda entretanto executá-lo.

python manage.py makemigrations

Veja a mensagem obtida.

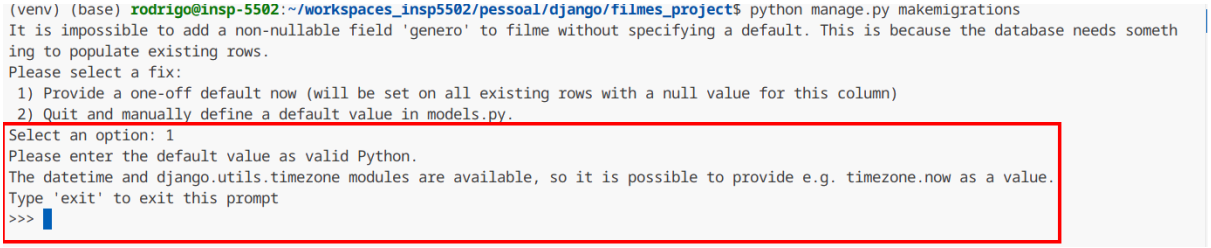


A sua aparição é natural. Como o relacionamento entre filmes e gêneros é obrigatório (não pode existir filme sem gênero), não podemos adicionar essa nova coluna à tabela de filmes, já que, no momento, ela já possui algumas linhas. Se a coluna fosse adicionada, qual valor ela teria? Temos algumas opções:

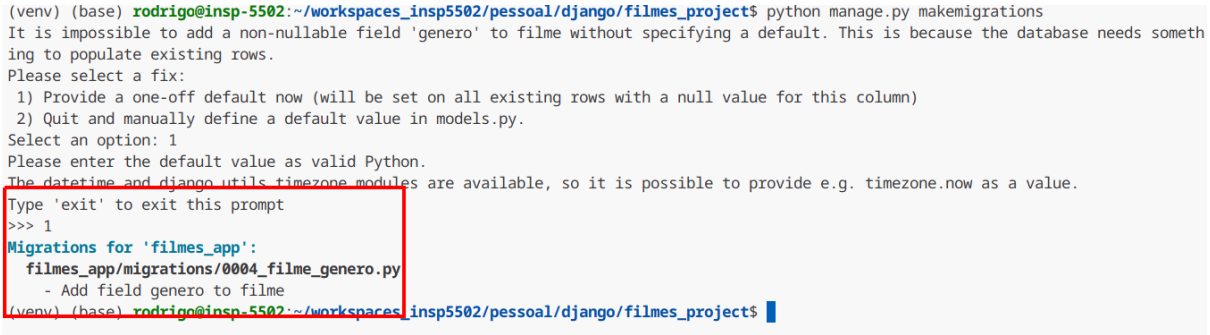
- Especificar agora, na linha de comando, um valor de id de gênero já existente, que será inserido para cada filme já existente.

- Apertar CTRL+C no terminal agora e ajustar um valor padrão no arquivo models.py.

Vamos optar pela primeira. Assim, digite 1. A seguir, você deverá digitar o valor desejado.



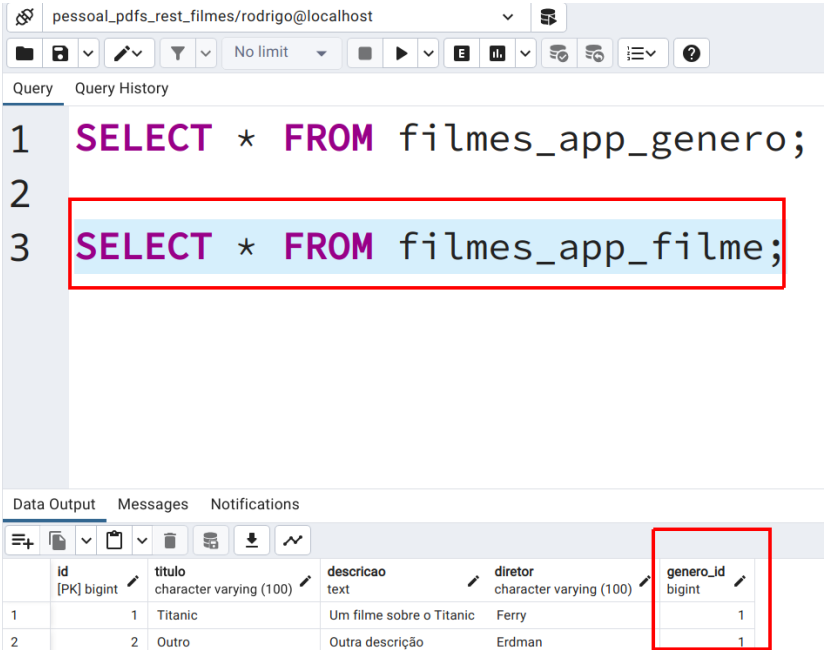
Basta digitar 1 novamente, que é o id do único gênero existente.



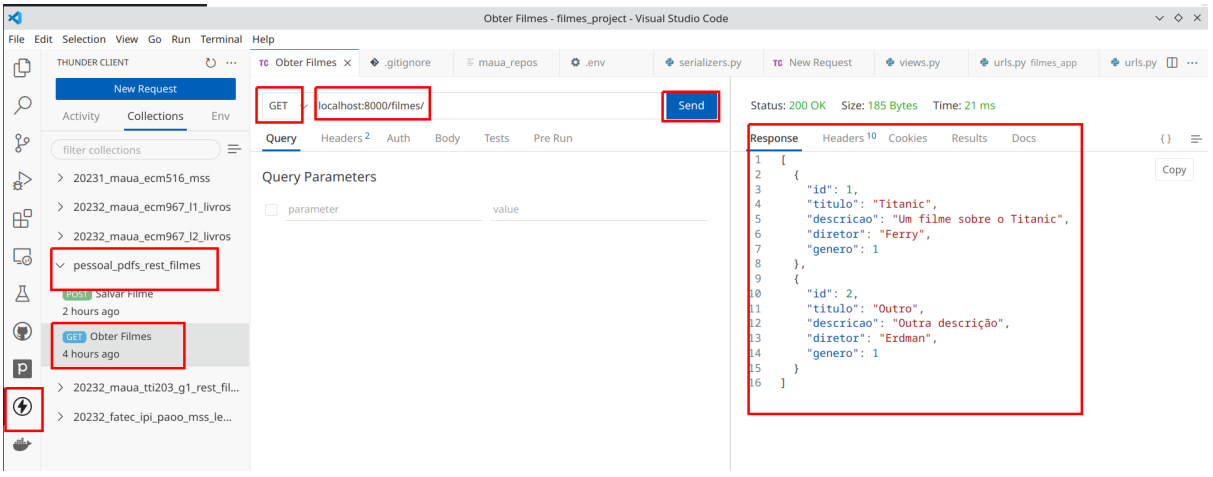
Agora podemos executar a nova migração.

```
python manage.py migrate
```

No pgAdmin, execute um SELECT na tabela de filmes e observe que a nova coluna existe. Cada filme existente tem o valor padrão associado a ele.



Na Thunder Client, faça uma requisição e veja o resultado.



Observe que cada filme possui apenas o id do gênero a que está associado. Podemos personalizar isso, fazendo com que cada filme possua o objeto JSON que representa seu gênero por completo.

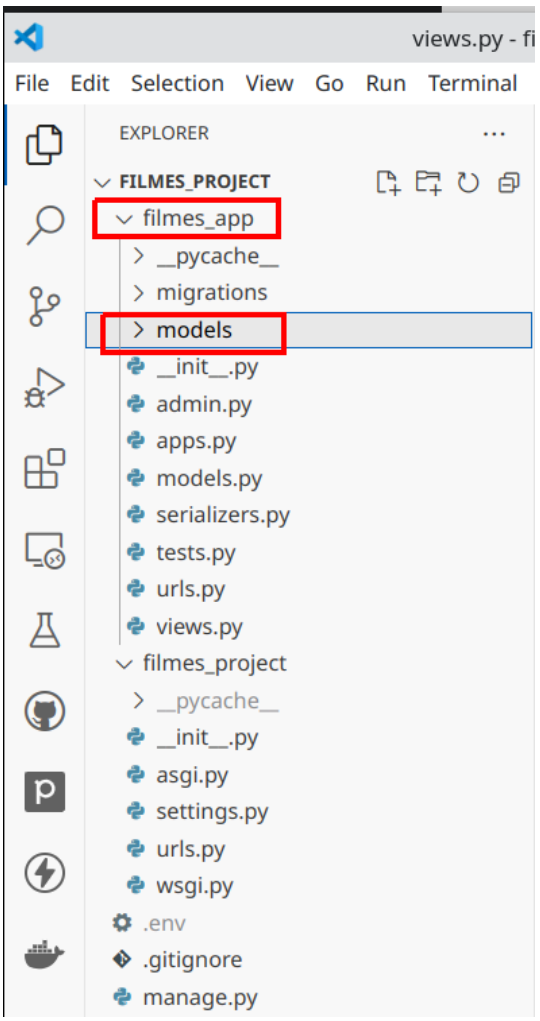
Para isso, no arquivo **serializers.py**, especifique uma classe serializadora para os gêneros, explicando que ela deve incluir todos os campos de gênero. Depois disso, na classe serializadora de filmes, explique que a forma a ser utilizada para serializar um gênero de um filme é aquela determinada pelo serializador de gênero, ou seja, incluindo todos os campos.

```
from rest_framework import serializers
from .models import Filme
from .models import Genero

class GeneroSerializer (serializers.ModelSerializer):
  class Meta:
    model = Genero
    fields = '__all__'

class FilmeSerializer (serializers.ModelSerializer):
  #o nome genero deve ser igual ao nome especificado no modelo de filme
  genero = GeneroSerializer()
  class Meta:
    model = Filme
    fields = '__all__'
```

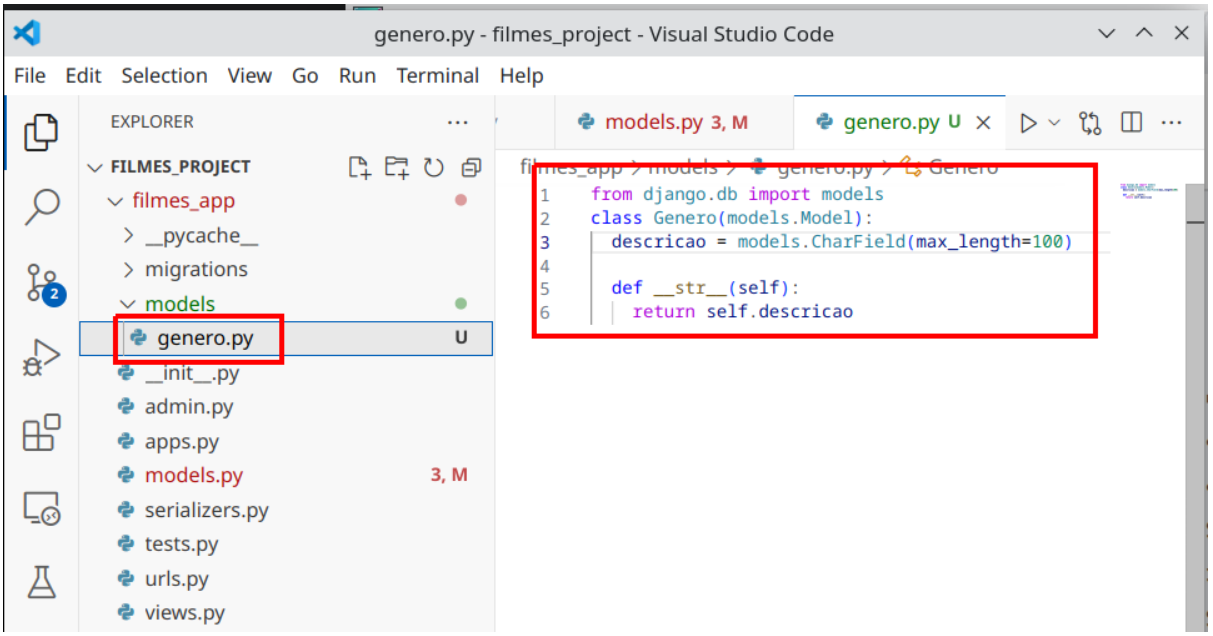
2.15 (Refatorando a aplicação: models, views e serializers) A medida que aplicação cresce, pode ser difícil de mantê-la caso tenhamos

- um único arquivo para abrigar as classes de modelo
- um único arquivo para abrigar as views
- um único arquivo para abrigar as classes serializadoras

Vamos organizar a estrutura da aplicação criando uma pasta para cada item desses e, dentro dela, um arquivo separado para cada classe. Começamos pelas classes de modelo. Crie uma pasta chamada **models** na raiz da aplicação.



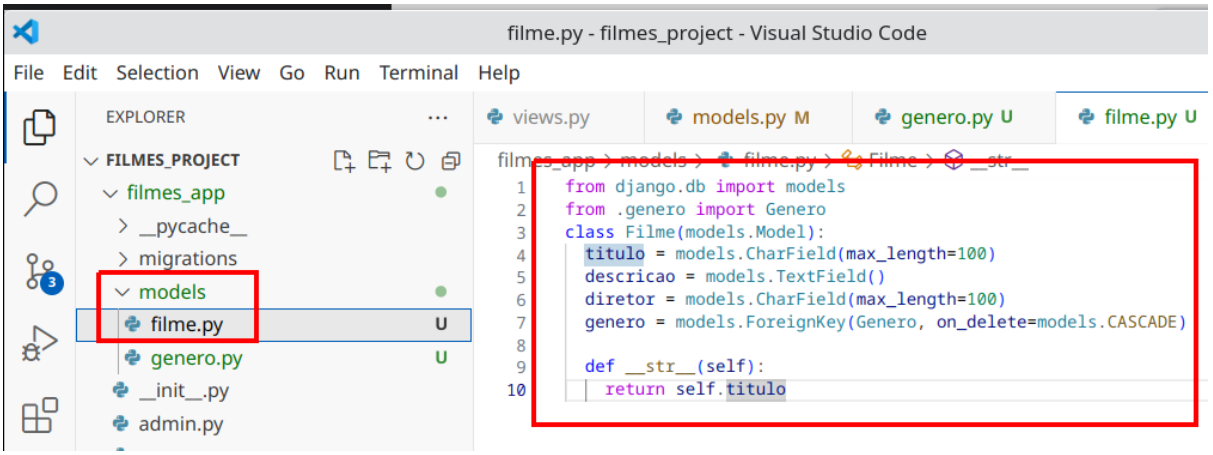
Na pasta **models**, crie um arquivo chamado **genero.py**. Ele servirá para definir a classe de modelo que descreve o que é um gênero.



O código do arquivo genero.py é o mesmo que tínhamos anteriormente no arquivo models.py.

```
from django.db import models
class Genero(models.Model):
  descricao = models.CharField(max_length=100)
  def __str__(self):
    return self.descricao
```

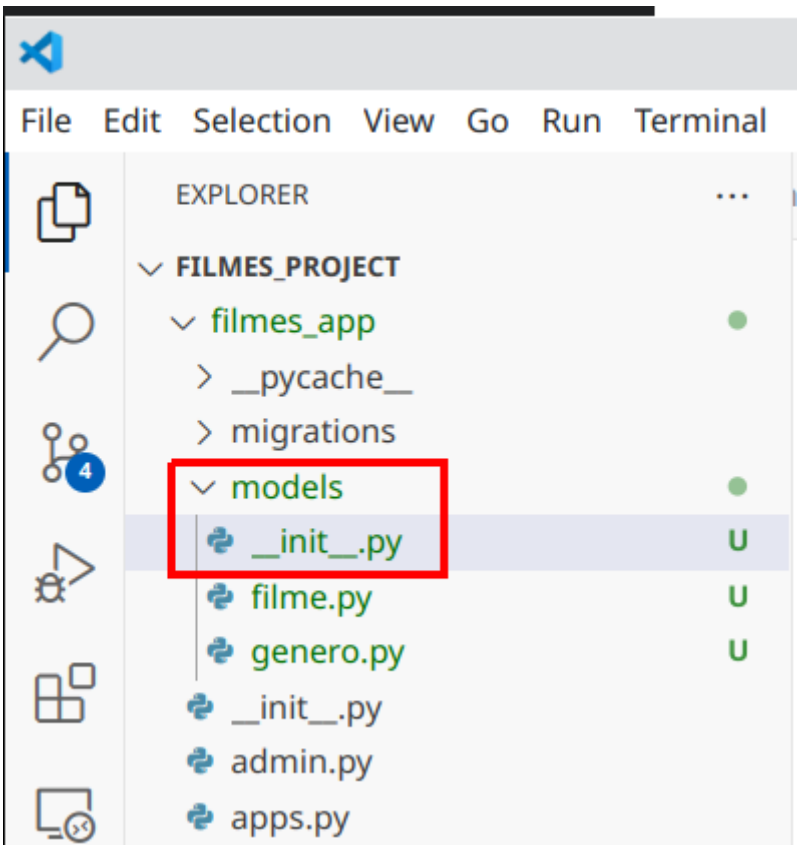
Repita os passos para a classe de modelo Filme, criando um arquivo **filme.py** para ela na pasta **models**. Observe que ela precisa importar o modelo de gênero também.



Veja o código do arquivo filme.py. É o mesmo que tínhamos antes também.

```
from django.db import models
from .genero import Genero
class Filme(models.Model):
  titulo = models.CharField(max_length=100)
  descricao = models.TextField()
  diretor = models.CharField(max_length=100)
  genero = models.ForeignKey(Genero, on_delete=models.CASCADE)

  def __str__(self):
    return self.titulo
```

Na pasta **models**, vamos criar um arquivo chamado **__init__.py** (dois underscores de cada lado). Ele serve para designar o diretório em que se encontra como um pacote Python, permitindo que módulos ali existentes sejam importados por outros. Ele também pode ter código de inicialização do pacote, entre outras coisas.

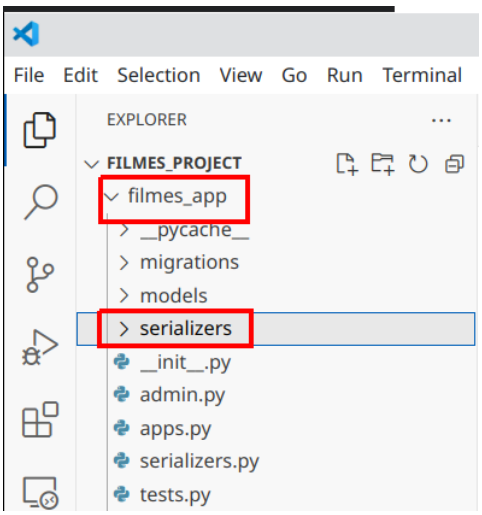


Neste arquivo, vamos importar nossas classes de modelo.

```
from .filme import Filme
from .genero import Genero
```

Observe que a existência do arquivo __init__.py numa pasta chamada models, nos permite continuar com os mesmos imports que antes faziam referência ao arquivo original **models.py**. Assim, você pode apagá-lo, ele não será mais necessário.

Agora vamos repetir o processo para as classes serializadoras. Comece criando uma pasta chamada **serializers** na raiz da sua aplicação.



Crie um arquivo chamado **genero_serializer.py** dentro da pasta recém criada. Veja seu conteúdo. Observe que, agora, os imports de modelos precisam ser relativos à existência das duas pastas. Não se esqueça de ajustar, indicando que a pasta models é subpasta de filmes_app. A definição do serializer é exatamente a mesma que tínhamos antes. Só o import mudou.

```
from rest_framework import serializers
#lembre-se de ajustar o import
from filmes_app.models import Genero

class GeneroSerializer (serializers.ModelSerializer):
  class Meta:
    model = Genero
    fields = '__all__'
```

Crie um arquivo chamado **filme_serializer.py** dentro da pasta recém criada. Veja seu conteúdo. Idêntico ao anterior também, a menos do import.

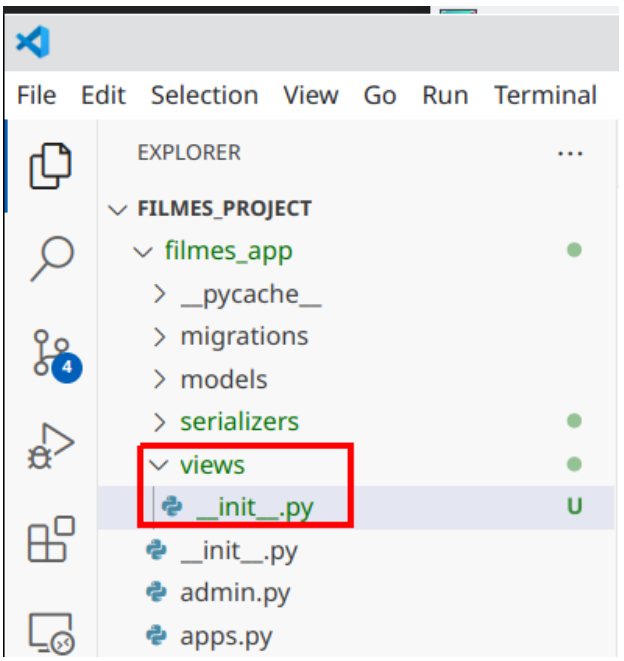
```
from rest_framework import serializers
#lembre-se de ajustar os imports
from filmes_app.models import Filme
from .genero_serializer import GeneroSerializer

class FilmeSerializer (serializers.ModelSerializer):
  #o nome genero deve ser igual ao nome especificado no modelo de filme
  genero = GeneroSerializer()
  class Meta:
    model = Filme
    fields = '__all__'
```

Crie também um arquivo chamado **__init__.py** na sua pasta serializers e importe a classe serializadora de filmes.

```
from .filme_serializer import FilmeSerializer
from .genero_serializer import GeneroSerializer
```

Neste momento, já podemos apagar o arquivo **serializers.py** original.

O processo para as views é análogo. Crie uma pasta chamada **views** na raiz da aplicação. Dentro dela, crie um arquivo chamado **__init__.py**.



A seguir, crie um arquivo chamado **filme_views.py**. Seu conteúdo é o mesmo que tínhamos anteriormente, a menos dos imports. Veja.

```
from rest_framework import generics
#ajuste os imports
from filmes_app.models import Filme
from filmes_app.serializers import FilmeSerializer

class FilmeListCreate(generics.ListCreateAPIView):
  queryset = Filme.objects.all()
  serializer_class = FilmeSerializer

class FilmeRetrieveDestroy(generics.RetrieveDestroyAPIView):
  queryset = Filme.objects.all()
  serializer_class = FilmeSerializer
```

No arquivo **__init__.py** da pasta recém criada, faça o import das views a partir do novo arquivo.

```
from .filme_views import FilmeListCreate, FilmeRetrieveDestroy
```

Neste momento, também já é possível apagar o arquivo **views.py** original.

Pode ser uma boa ideia reiniciar o servidor e realizar novos testes.

Professor Rodrigo Bossini <professorbossini@gmail.com>
Produção: setembro de 2023