

1 Introdução

Django é um framework “batteries-included”. Ele traz diversas funcionalidades comuns prontas para uso. Uma delas é uma aplicação capaz de lidar com autenticação e autorização. Veja as suas principais características.

Modelos Predefinidos: Inclui uma classe de modelo que representa **usuários**, contendo muitos dos campos comuns necessários, como username, password e email.

Formulários de Autenticação: Inclui templates HTML básicos para as operações mais elementares como login, logout e troca de senha, os quais podem ser personalizados.

Views e URLs associadas: Complementando os formulários, o Django já possui views (funções ou classes que determinam o que mostrar ao usuário) e URLs relacionadas para processar pedidos de autenticação. Por exemplo, há views para fazer login, fazer logout, mudar a senha e assim por diante.

Sistema de Permissões: Além da autenticação básica, o Django oferece um sistema de autorização que permite definir permissões específicas para diferentes tipos de usuários.

Segurança: As senhas são armazenadas criptografadas. Além disso, ele oferece proteção contra ataques de força bruta (tentativas repetidas de adivinhar uma senha, por exemplo).

Neste material, veremos como utilizar a aplicação com

- **api REST com o Django Rest Framework (DRF)**
- **templates (páginas HTML já pré definidas)**

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 instalado, 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.

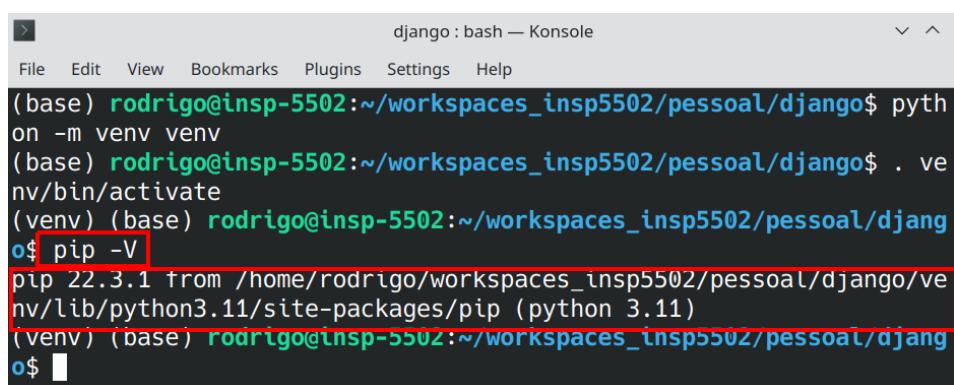
Depois da criação do ambiente virtual, é preciso ativá-lo. Usuários Windows, podem utilizar um terminal “cmd” ou um terminal “Powershell”. A tabela a seguir resume a forma como o ambiente virtual Python pode ser ativado em qualquer caso.

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

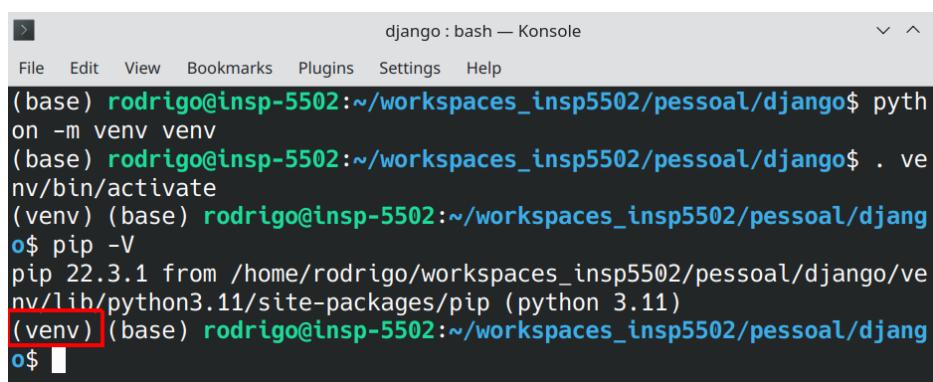
pip -V

A pasta de seu ambiente virtual deve ser exibida.



```
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.

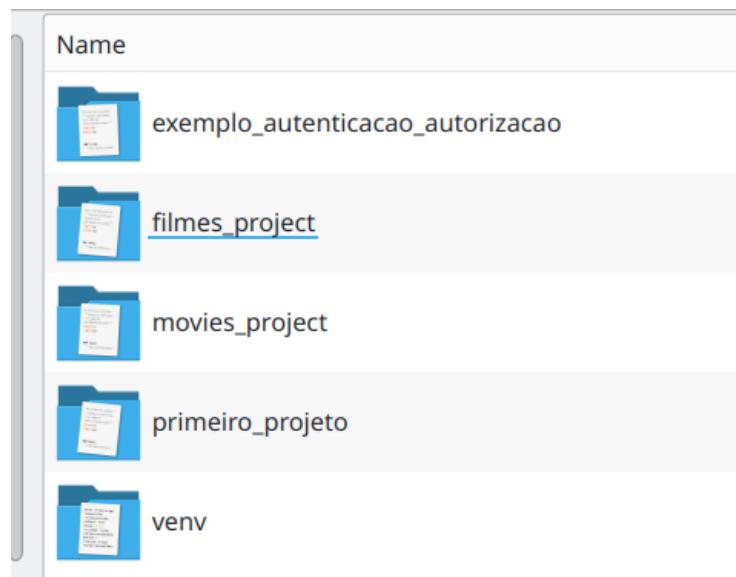
O próximo passo é criar um **projeto Django**. Um projeto Django é uma coleção de aplicações Django e configurações. Em geral, ele possui

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

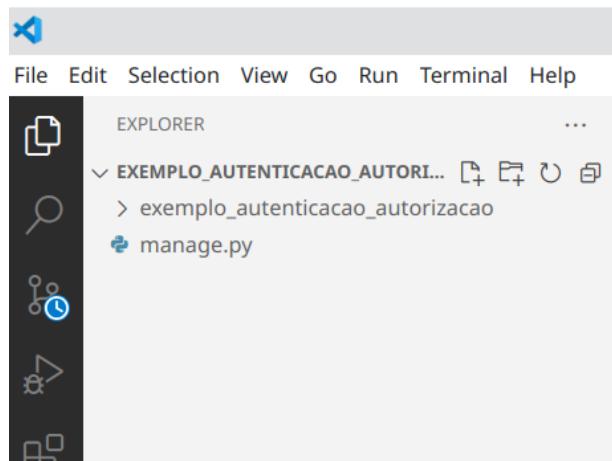
Podemos criar o projeto com

```
django-admin startproject exemplo_autenticacao_autorizacao
```

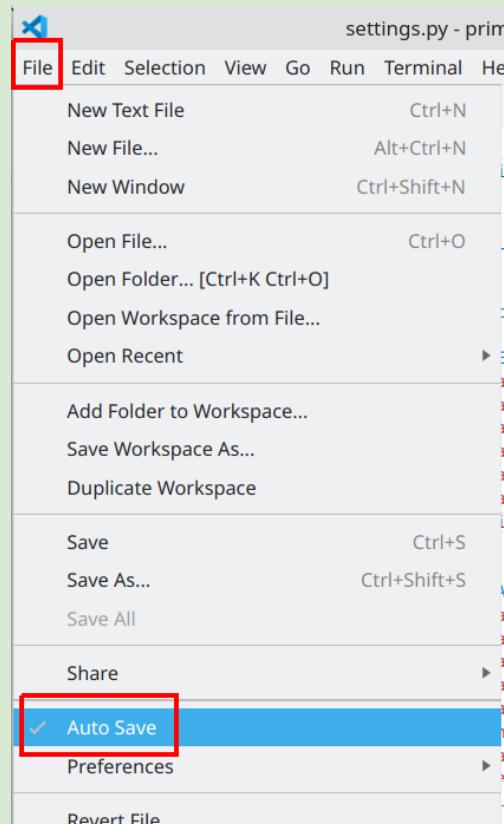
Uma pasta deve ter sido criada, ao lado da pasta que representa o seu ambiente virtual.



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.

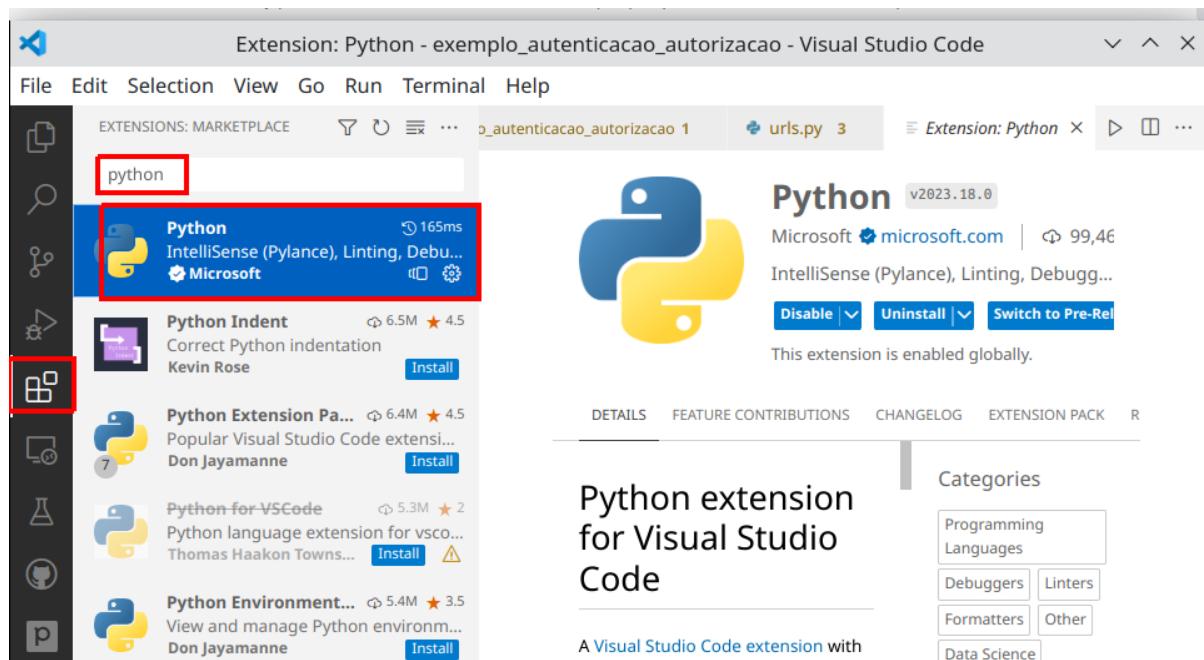
Cuidado. O ambiente virtual se encontra numa pasta chamada **venv** que está um nível acima da pasta em que estamos no momento. Por isso, use `../` para acessá-la. Em seu ambiente, é possível que ela esteja num diretório diferente. Se tiver dúvidas, use seu path completo. Por exemplo: `C:\Users\usuario\Documents\dev\django\venv` se estiver no Windows.

Veja o resultado esperado.



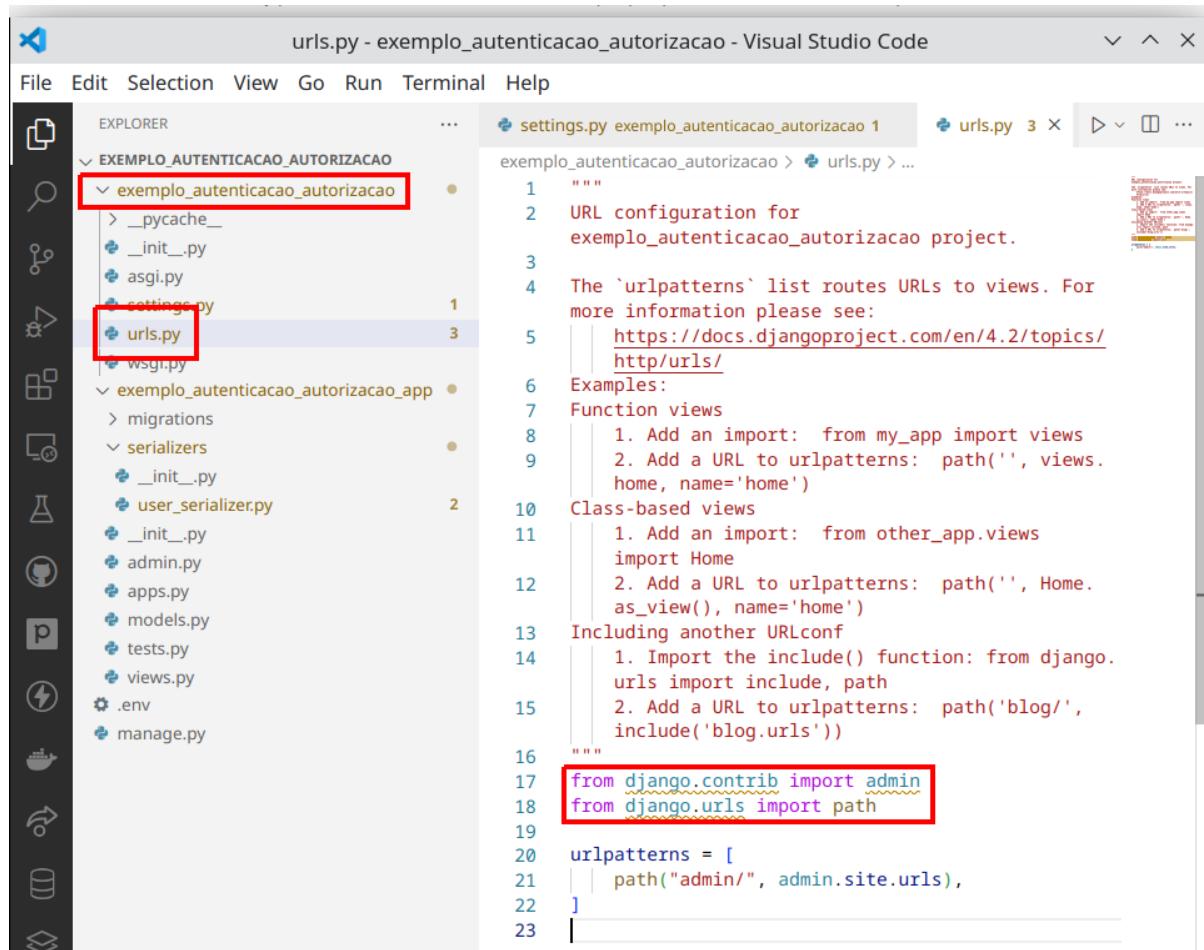
Use o comando apropriado para o seu sistema operacional.

Caso ainda não tenha feito, instale a extensão Python no VS Code.



A extensão Python para o VS Code inclui diversas outras. Uma delas se chama **PyLance**. Ela oferece dicas para completar código, permite a “navegação no código” (clicar num nome e ir

até a sua definição) entre outras coisas A PyLance baseia o seu funcionamento no ambiente Python ativo no momento. Embora tenhamos ativado o ambiente que inclui o Django em nossos terminais, pode ser que o VS Code esteja utilizando outro ambiente. Se for o caso, A PyLance não será capaz de dar dicas envolvendo os pacotes específicos do nosso ambiente virtual. Abra, por exemplo, o arquivo **urls.py** do projeto. É possível que os imports do Django apareçam sublinhados em amarelo.

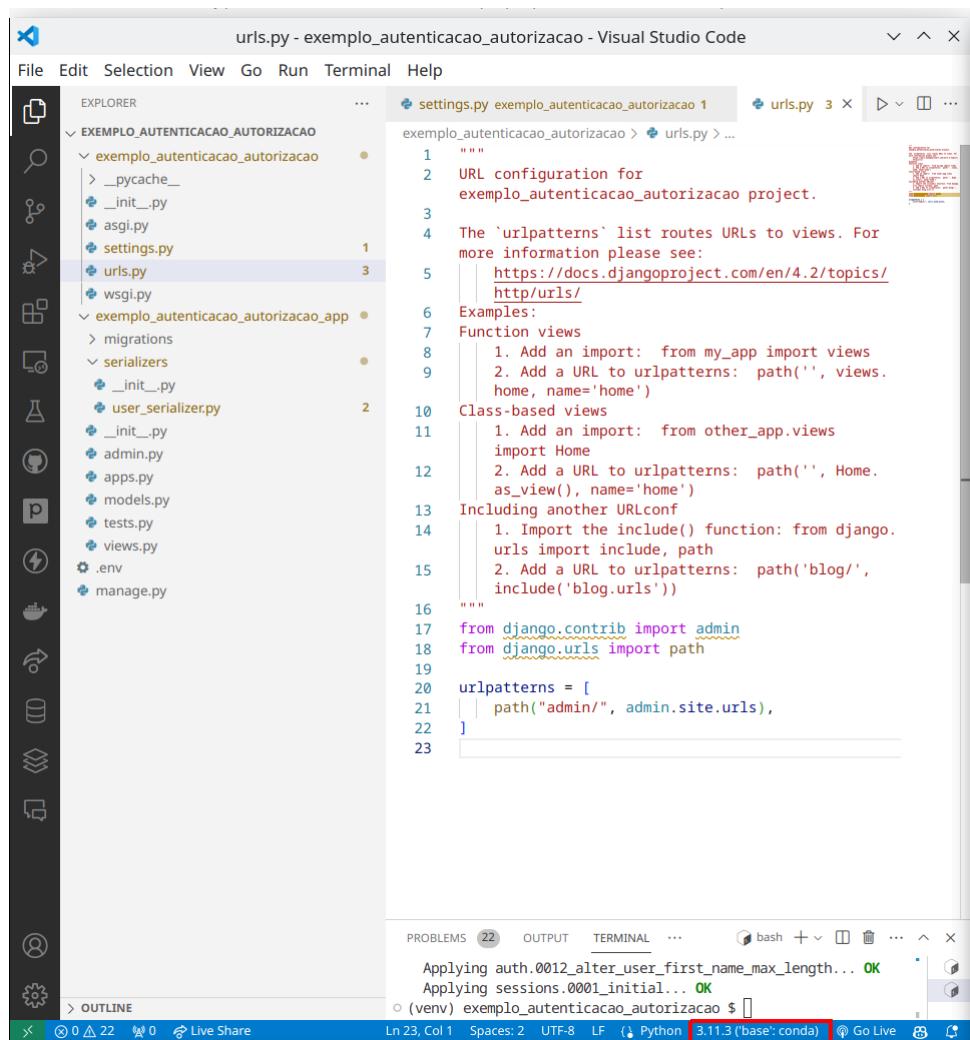


```

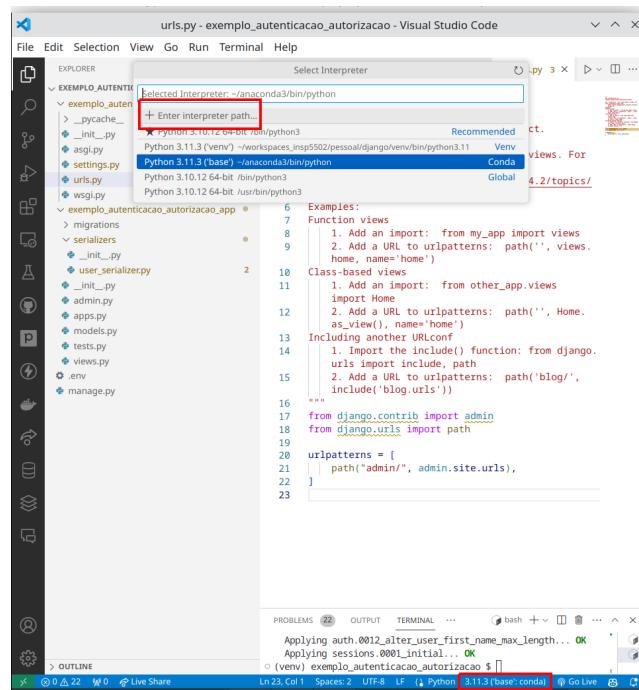
1 """
2 URL configuration for
3     exemplo_autenticacao_autorizacao project.
4
5 The `urlpatterns` list routes URLs to views. For
6 more information please see:
7     https://docs.djangoproject.com/en/4.2/topics/
8     http/urls/
9 Examples:
10 Function views
11     1. Add an import: from my_app import views
12     2. Add a URL to urlpatterns: path('', views.
13         home, name='home')
14 Class-based views
15     1. Add an import: from other_app.views
16         import Home
17     2. Add a URL to urlpatterns: path('', Home.
18         as_view(), name='home')
19 Including another URLconf
20     1. Import the include() function: from django.
21         urls import include, path
22     2. Add a URL to urlpatterns: path('blog/',
23         include('blog.urls'))
24
25 from django.contrib import admin
26 from django.urls import path
27
28 urlpatterns = [
29     path("admin/", admin.site.urls),
30 ]

```

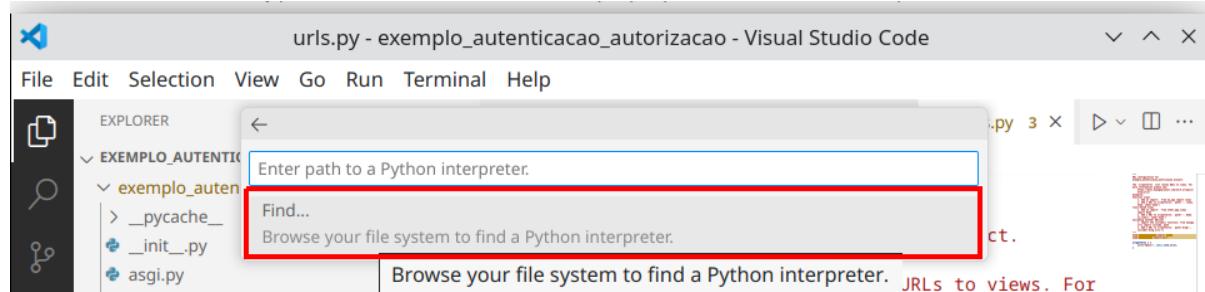
Se for o caso, você pode trocar o ambiente Python utilizado pelo VS Code. Ele aparece na barra azul de status, na parte inferior do VS Code. No exemplo a seguir, não estamos usando o venv.



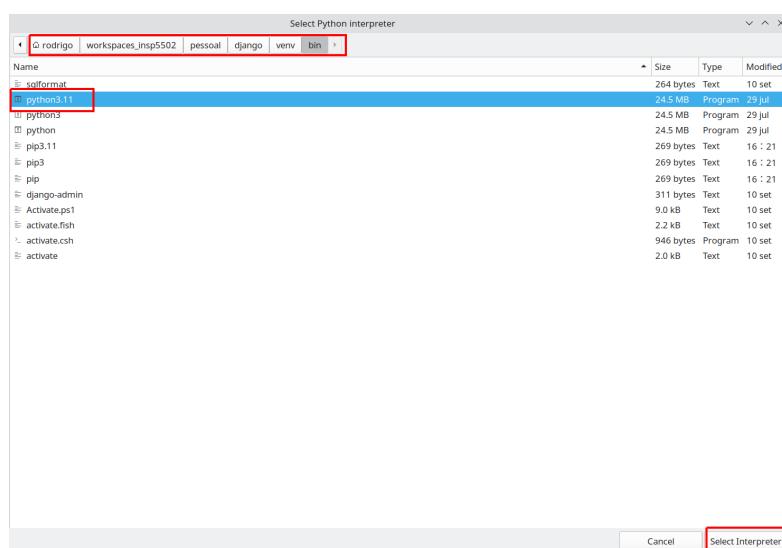
Clique nesta região. Você poderá escolher o seu ambiente.



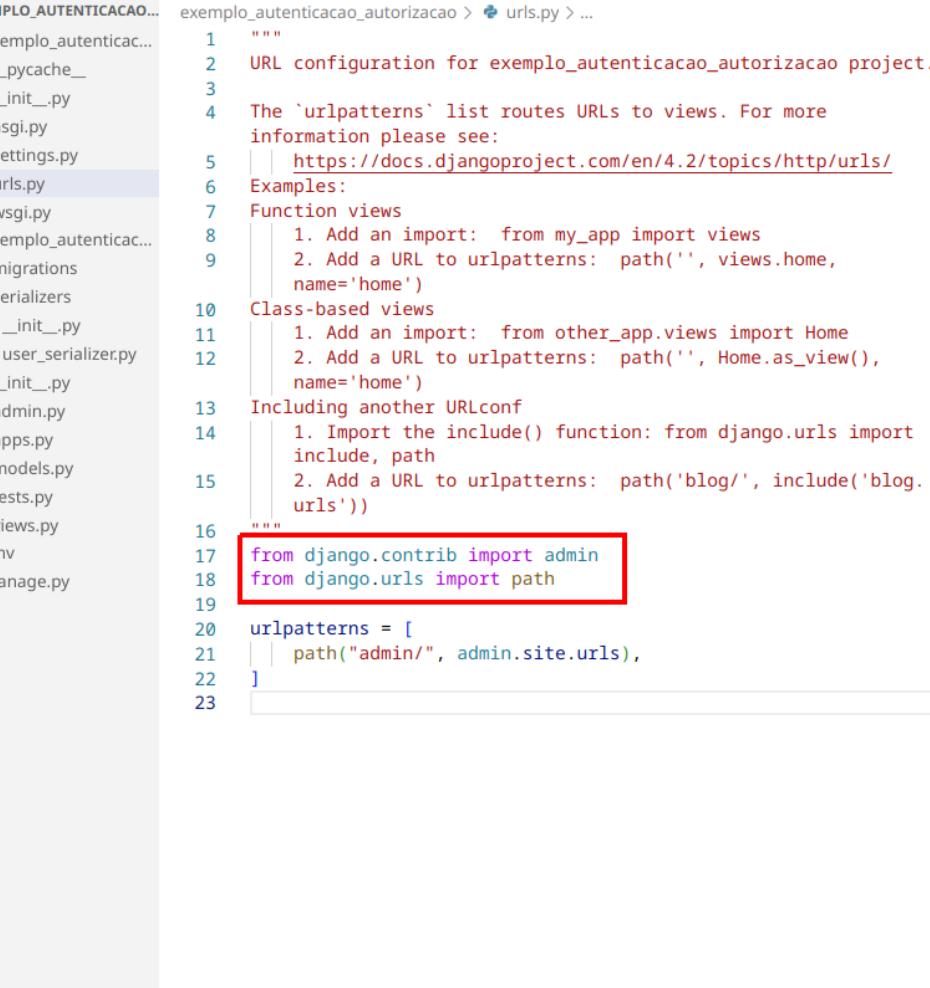
Clique em Find.



Navegue no seu sistema de arquivos para encontrar o executável Python de seu ambiente. Observe que ele fica na sua pasta venv. A sua versão pode ser diferente.



Veja o resultado depois da troca.



```
1 """
2 URL configuration for exemplo_autenticacao_autorizacao project.
3
4 The `urlpatterns` list routes URLs to views. For more
5 information please see:
6     https://docs.djangoproject.com/en/4.2/topics/http/urls/
7 Examples:
8     Function views
9         1. Add an import: from my_app import views
10            2. Add a URL to urlpatterns: path('', views.home,
11               name='home')
12
13 Class-based views
14     1. Add an import: from other_app.views import Home
15            2. Add a URL to urlpatterns: path('', Home.as_view(),
16               name='home')
17
18 Including another URLconf
19     1. Import the include() function: from django.urls import
20        include, path
21     2. Add a URL to urlpatterns: path('blog/', include('blog.
22           urls'))
23
24 """
25
26 from django.contrib import admin
27 from django.urls import path
28
29
30 urlpatterns = [
31     path("admin/", admin.site.urls),
32 ]
```

2.2 (Novas dependências: Django Rest Framework (DRF) e psycopg2 (driver PostgreSQL))

Como comentamos, as funcionalidades de autenticação/autorização serão oferecidas por meio de

- templates
- API REST

Por conta da API Rest, vamos instalar o Django Rest Framework(DRF). Veja o site do DRF.

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

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

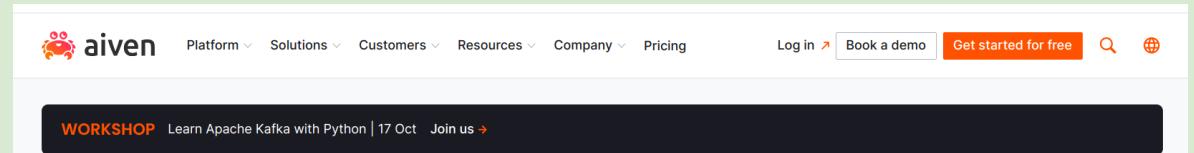
`pip install djangorestframework psycopg2`

para instalar ambos.

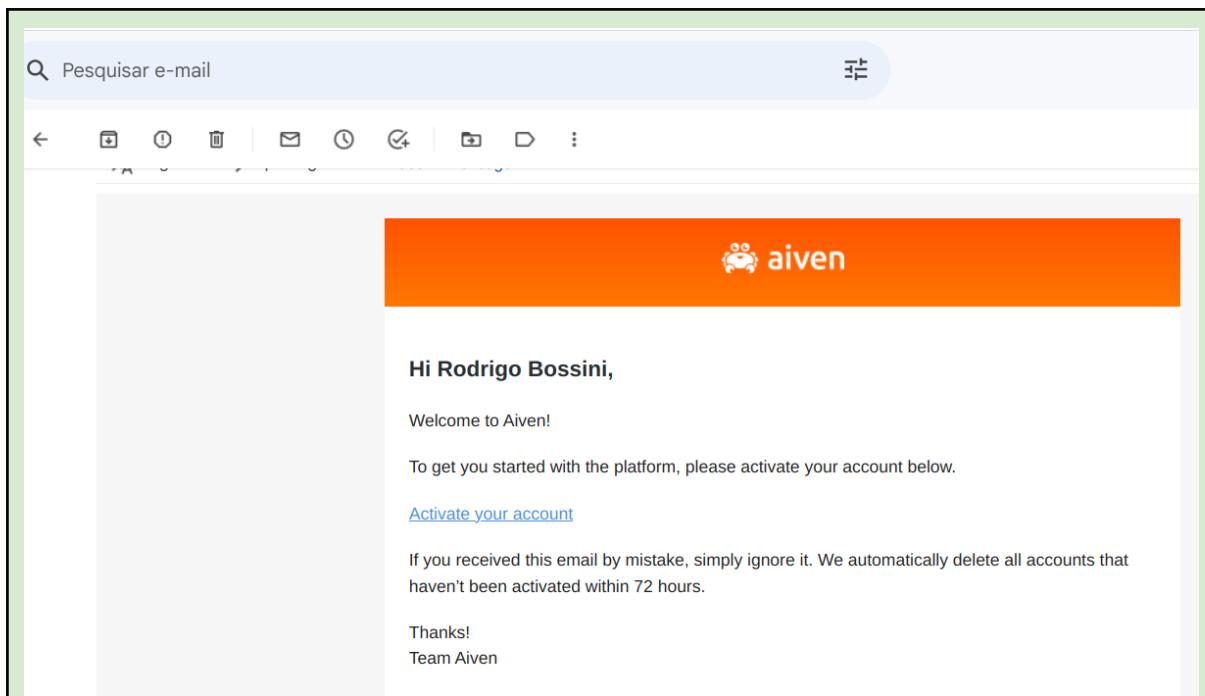
Nota. Talvez você queira fazer uso de um serviço de computação em nuvem, que ofereça uma instância do PostgreSQL gratuitamente. Uma opção é o **aiven**. Veja sua página oficial.

<https://aiven.io/>

Clique em Get Started for Free.



Depois de criar a conta, você receberá um e-mail de confirmação. É preciso realizar esta confirmação antes de utilizar o ambiente.



Depois de fazer login, você precisa criar um serviço.

Home Projects Billing Support Admin My Organization

PROJECT professorbossini My Organization / professorbossini / Services

Services

Search services by name, plan, cloud and tags... Filter list

Show only services with alerts

Create example data pipeline

Combine multiple services into a data pipeline with this example guide.

Create example data pipeline

Escolha o PostgreSQL.

Create new service My Organization / professorbossini / Select service

Select service

 PostgreSQL® PostgreSQL - Object-Relational Database Management System	 MySQL MySQL - Relational Database Management System	 Redis™* Redis - In-Memory Data Structure Store
 Apache Kafka® Kafka - High-Throughput Distributed Messaging System	 OpenSearch® OpenSearch - Search & Analyze Data in Real Time, derived from Elasticsearch v7.10.2	 Apache Cassandra® Cassandra - Distributed NoSQL data store
 InfluxDB® InfluxDB - Distributed Time Series Database	 Grafana® Grafana - Metrics Dashboard	 M3DB M3DB - Distributed time series database
 M3 Aggregator M3 Aggregator - Aggregates metrics and provides downsampling	 ClickHouse® ClickHouse - Column-oriented DBMS for online analytical processing	 Apache Flink® Flink - Stateful Computations over Data Streams

Escolha o plano grátis e clique para criar um serviço. Se desejar, na parte inferior da tela, altere o nome do serviço. Use um nome que te ajude a lembrar a razão de ser do serviço, seu propósito, sua finalidade.

Create new service My Organization / professorbossini / Select service / PostgreSQL®

Version 15 ▾

Service summary

Name pg-1bd68abd

Service PostgreSQL 15

Cloud AWS Amazon Web Services

Region Asia, India - Amazon Web Services: Mumbai

Plan Free-1-5gb

② 2 CPU ② 1 GB RAM ② 5 GB storage ② backups for disaster recovery ② 1 node

Monthly price **\$0 USD**

Create free service

1. Select service region

Asia Pacific Europe North America

aws-ap-south-1

② India - Amazon Web Services: Mumbai

2. Select service plan

To get started with higher plan types, select full platform. Please refer to the [plan comparison](#) for more information.

Free-1-5gb \$0 / month

② 2 CPU ② 1 GB RAM ② 5 GB storage ② backups for disaster recovery ② 1 node

3. Provide service name

The service name cannot be changed afterwards.

Name* pg-1bd68abd

No próximo passo, você pode adicionar os endereços IP a partir dos quais seu serviço aceitará conexões. É uma medida de segurança. No momento, não precisamos nos preocupar com isso. Clique Next.

Allowed inbound IP addresses

Your service is open to all IP ranges.

Improve network security by restricting access to trusted IP addresses. Bulk upload

IP address range (CIDR block)	Comment
Add IP address	Add optional comment
0.0.0.0/0	0 / 1024

SERVICE
PostgreSQL 15
Asia, India - Amazon Web Services: Mumbai

DEPLOYMENT STATUS
Nodes 1
Rebuilding

[Go to service overview](#)

A seguir, você também pode adicionar dados de bases de dados que eventualmente já possua. Também não estamos interessados no momento. Clique Next.

Add data to your database

Using the Aiven CLI, you can add sample data or migrate data to your Aiven for PostgreSQL® database. To keep your database

Migrate database	Tutorial	Test data
Set up migration Go to instructions	Learn how to migrate to Aiven for PostgreSQL® using Aiven CLI Migrate your data by following the step-by-step instructions in our documentation. Go to instructions	Pagila sample data Size ~5 MB Import the sample database Pagila into your PostgreSQL service. Load sample data Go to instructions

SERVICE
PostgreSQL 15
Asia, India - Amazon Web Services: Mumbai

DEPLOYMENT STATUS
Nodes 1
Rebuilding

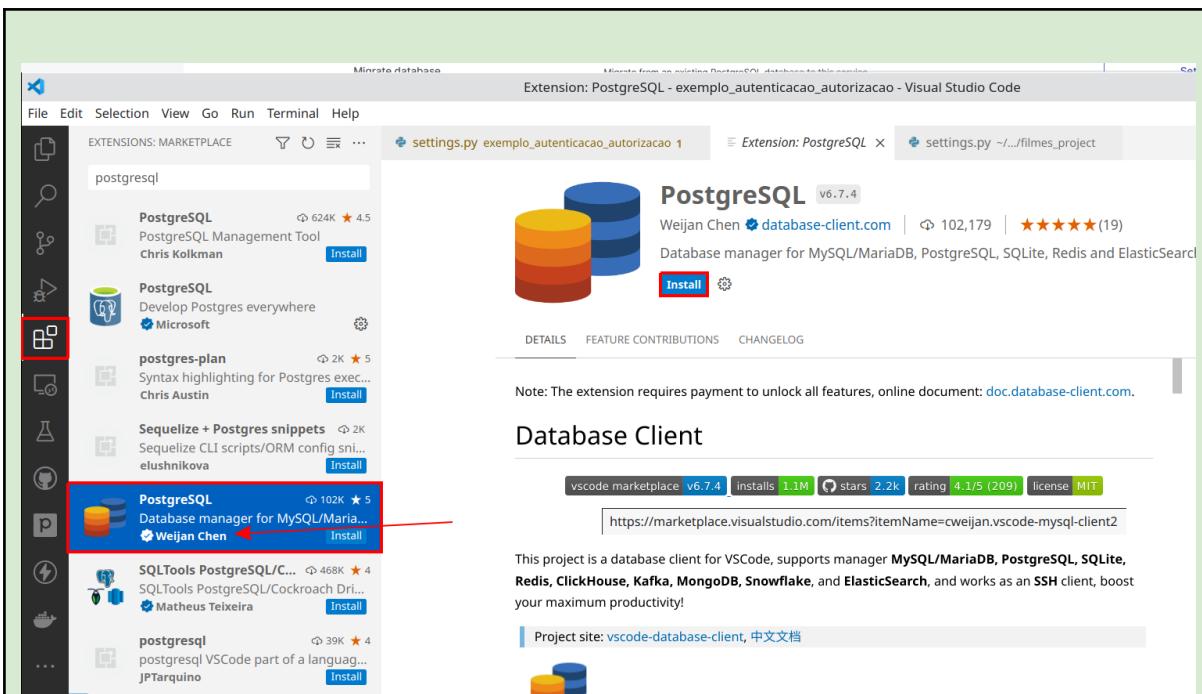
[Go to service overview](#)

Depois disso, você pode escolher extensões que eventualmente deseje instalar.

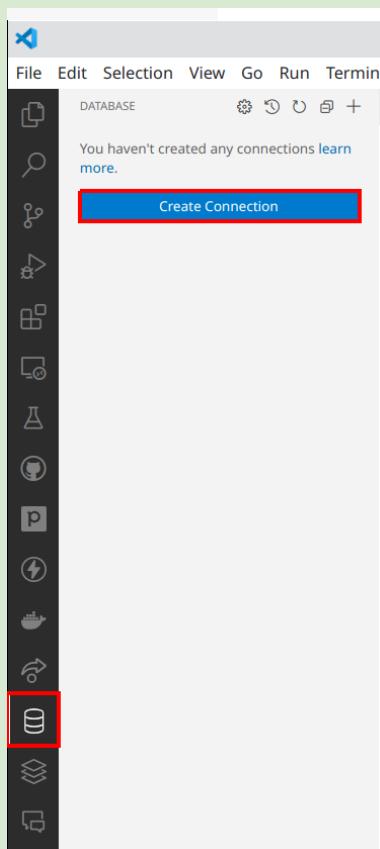
Apenas instale aquelas de que realmente precisa. Se você estiver na dúvida, não instale nenhuma, por enquanto. Clique para finalizar quando terminar.

Observe que, agora, você pode obter os dados para acesso à sua base de dados remota.

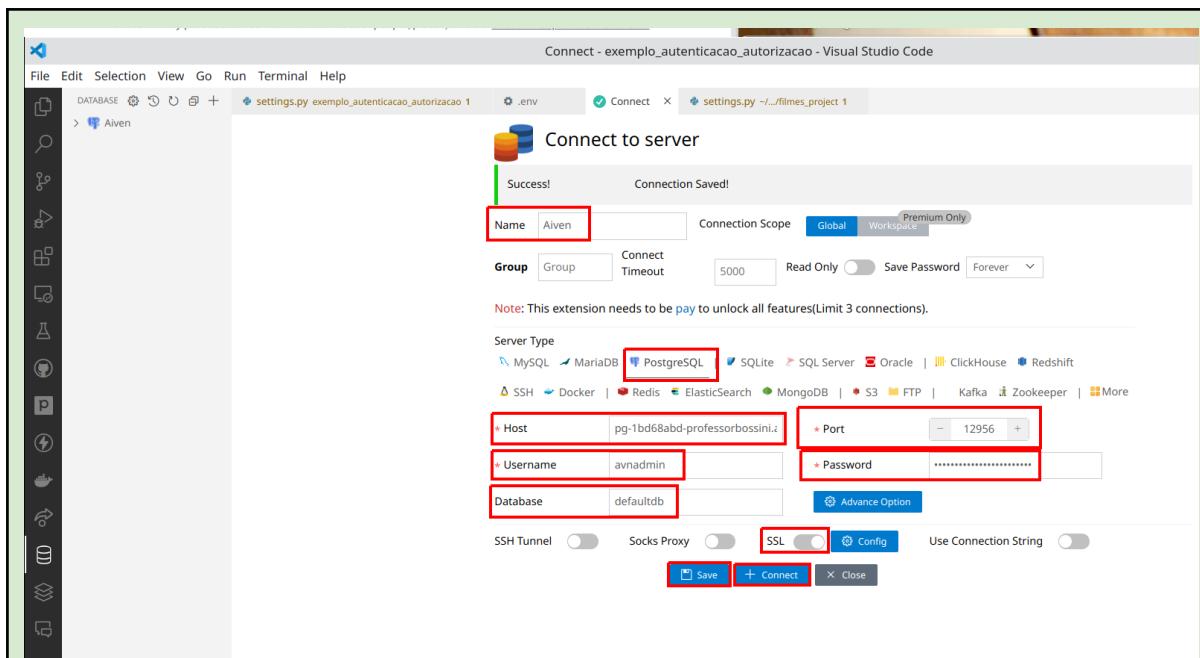
Você pode utilizar diferentes clientes para se conectar à sua base. Neste exemplo, vamos utilizar a seguinte extensão do VS Code.



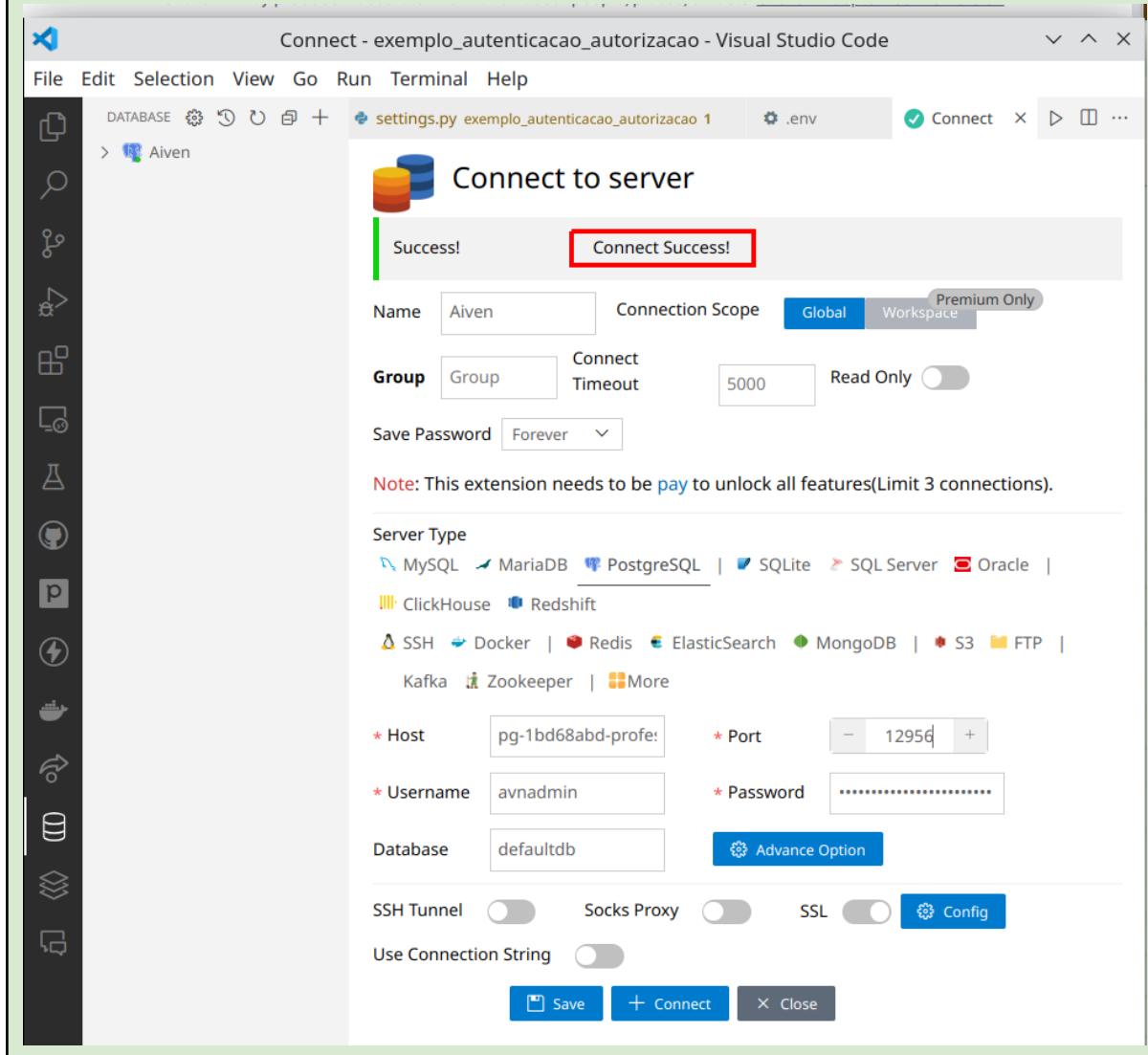
Depois da instalação, você pode criar uma conexão.



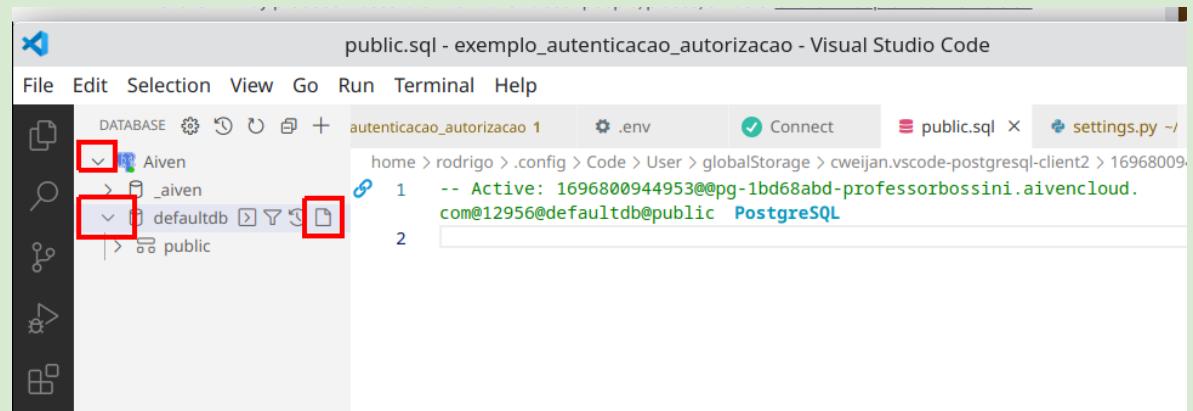
A seguir, preencha os campos, clique em **Save** e **Connect**.



Se tudo deu certo, você deverá ver a mensagem **Connect success.**



Agora é preciso expandir a conexão e o banco de dados, além de clicar no ícone destacado a seguir.

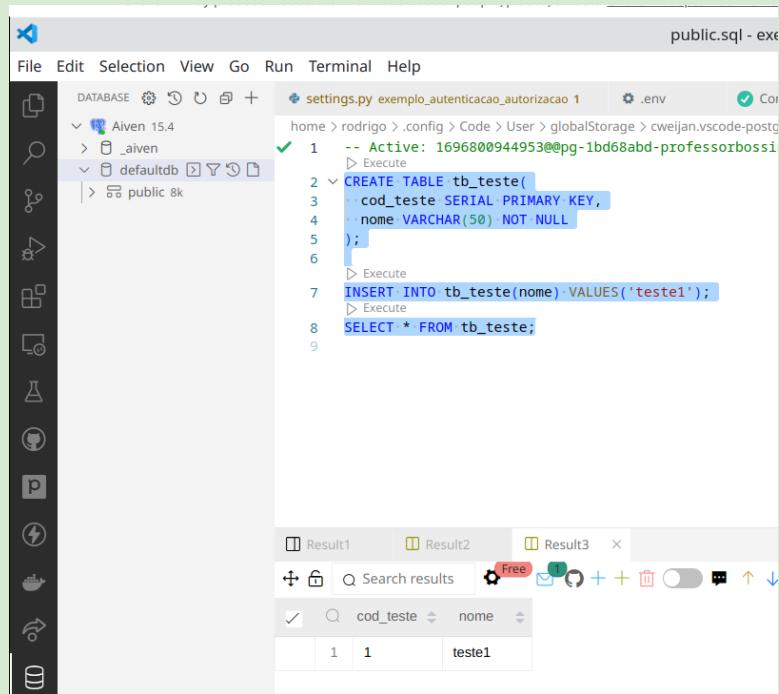


Você terá acesso a um editor em que poderá digitar seus comandos SQL. Faça o seguinte teste.

```
CREATE TABLE tb_teste(
cod_teste SERIAL PRIMARY KEY,
nome VARCHAR(50) NOT NULL
);

INSERT INTO tb_teste(nome) VALUES ('teste1');
SELECT * FROM tb_teste;
```

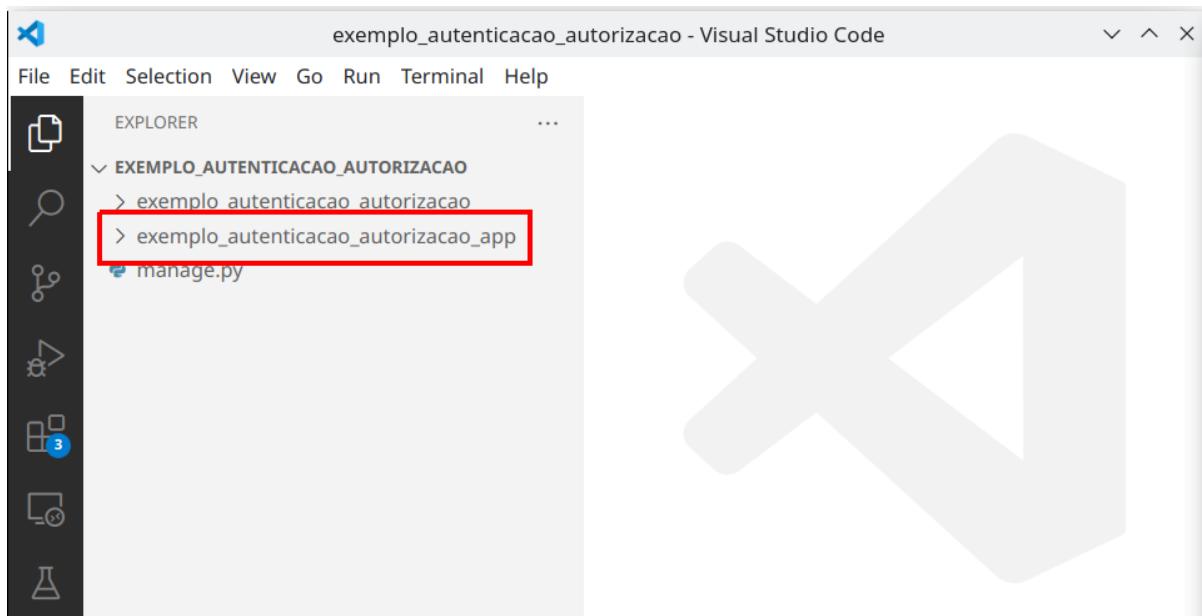
Selecione o código e aperte CTRL + Enter para executar.



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 exemplo_autenticacao_autorizacao_app
```

Veja o resultado esperado.



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",
]
```

```

"exemplo_autenticacao_autorizacao"
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",

```

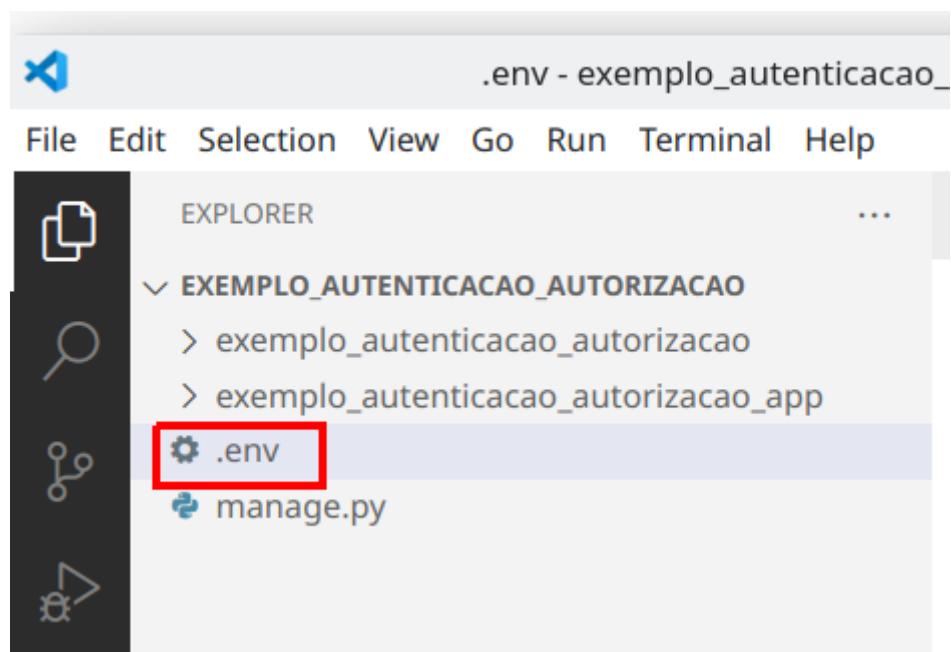
2.5 (Isolando os dados de acesso à base e adicionando eles ao arquivo settings.py com o mecanismo dotenv) Desejamos isolar os dados de acesso à base de dados por duas razões, pelo menos.

- controle de versão em repositórios públicos
- facilidade de alteração dos valores em função do ambiente (desenvolvimento, produção etc)

Para defini-los, vamos usar o pacote **django-environ**. Faça a sua instalação com

```
pip install django-environ
```

A seguir, crie um arquivo chamado **.env**, na raiz do projeto, fora de qualquer pasta, lado a lado com o arquivo **manage.py**.



Veja seu conteúdo. Neste exemplo, estamos utilizando um banco de dados gerenciado pelo PostgreSQL por meio do serviço Aiven. Você também pode usar uma base local, se desejar.

Nota. O valor de SECRET_KEY, neste exemplo, foi obtido do próprio arquivo settings.py criado originalmente quando criamos o projeto. Basta copiar seu conteúdo.

```
SECRET_KEY=django-insecure-es!1$*fjkcfm7j_c=tyy-it9r#w9+eh6f0v2by80cp
i6jf4^u!
DEBUG=True
DATABASE_DEFAULT_NAME=defaultdb
DATABASE_DEFAULT_USER=avnadmin
DATABASE_DEFAULT_PASSWORD=sua senha do Aiven aqui
DATABASE_DEFAULT_HOST=pg-1bd68abd-professorbossini.aivencloud.com
DATABASE_DEFAULT_PORT=12956
```

Já no arquivo **settings.py** do projeto, faça a leitura do seu arquivo .env e, então, passe a utilizar os valores nele definidos, por meio da função env.

```
...
from pathlib import Path
import environ
env = environ.Env(
    #deixamos False por padrão
    #caso o .env não defina, por segurança, é melhor, já que o ambiente
    #pode ser o de produção
    DEBUG = (bool, False)
)

# 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')

ALLOWED_HOSTS = []
# Application definition
INSTALLED_APPS = [
```

```

...
# Database
# https://docs.djangoproject.com/en/4.2/ref/settings/#databases

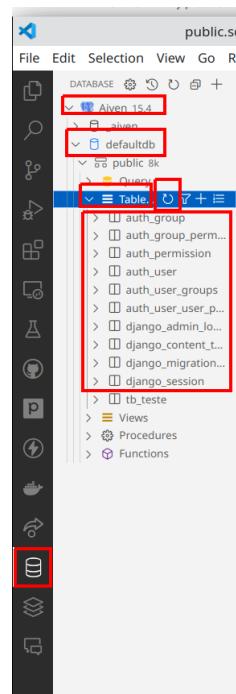
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': env('DATABASE_DEFAULT_NAME'),
        'USER': env('DATABASE_DEFAULT_USER'),
        'PASSWORD': env('DATABASE_DEFAULT_PASSWORD'),
        'HOST': env('DATABASE_DEFAULT_HOST'),
        'PORT': env('DATABASE_DEFAULT_PORT'),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

```

Faça uma primeira execução de migração, atualizando as bases referentes às aplicações já existentes no projeto.

`python manage.py migrate`

Se tudo deu certo, você deverá visualizar as tabelas criadas pelo Django.



2.6 (Usando a classe de modelo User do Django para fazer cadastro de novos usuários) A aplicação de autenticação/autorização do Django possui uma classe de modelo chamada User. Como o nome sugere, ela serve para representar possíveis usuários do sistema. Se desejar, você pode averiguar seu código fonte. Ele pode ser encontrado em seu diretório venv. Algo como

venv/lib/python3.11/site-packages/django/contrib/auth/

Se desejar, você também pode encontrar o código fonte no Github.

<https://github.com/django/django/blob/main/django/contrib/auth/models.py>

Veja um trecho da classe User.

```
403
404
405 class User(AbstractUser):
406     """
407     Users within the Django authentication system are represented by this
408     model.
409
410     Username and password are required. Other fields are optional.
411     """
412
413     class Meta(AbstractUser.Meta):
414         swappable = "AUTH_USER_MODEL"
415
```

Observe como ela herda de AbstractUser. Veja um trecho dela.

```
334     class AbstractUser(AbstractBaseUser, PermissionsMixin):
335         """
336         An abstract base class implementing a fully featured User model with
337         admin-compliant permissions.
338
339         Username and password are required. Other fields are optional.
340         """
341
342         username_validator = UnicodeUsernameValidator()
343
344         username = models.CharField(
345             _("username"),
346             max_length=150,
347             unique=True,
348             help_text=_(
349                 "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only."
350             ),
351             validators=[username_validator],
352             error_messages={
353                 "unique": _("A user with that username already exists."),
354             },
355         )
356         first_name = models.CharField(_("first name"), max_length=150, blank=True)
357         last_name = models.CharField(_("last name"), max_length=150, blank=True)
358         email = models.EmailField(_("email address"), blank=True)
359         is_staff = models.BooleanField(
360             _("staff status"),
361             default=False,
362             help_text=_("Designates whether the user can log into this admin site."),
363         )
364         is_active = models.BooleanField(
365             _("active"),
366             default=True,
367             help_text=_(
368                 "Designates whether this user should be treated as active. "
369                 "Unselect this instead of deleting accounts."
370             ),
371         )
```

Veja uma descrição de alguns campos que a classe User possui.

username: O nome de usuário, único, usado para autenticar o usuário. É um campo obrigatório.

password: Senha do usuário, criptografada pelo Django antes de ser armazenada.

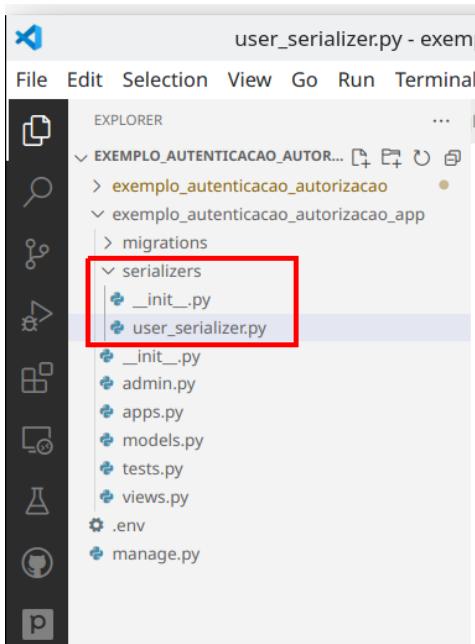
email: Endereço de email do usuário.

first_name e **last_name**: Primeiro e último nome do usuário.

groups: Um relacionamento muitos para muitos (ManyToManyField) com o modelo Group, permitindo que você agrupe usuários e atribua permissões a esses grupos.

user_permissions: Um relacionamento muitos para muitos (ManyToManyField) com o modelo Permission, permitindo que você atribua permissões específicas a um usuário específico, independentemente dos grupos.

Como a classe de modelo já está pronta, vamos criar uma classe **serializadora** para escolher os campos que nos são de interesse. Para isso, **crie uma pasta chamada serializers na raiz da aplicação e, dentro dela, um arquivo chamado __init__.py, vazio num primeiro momento. Crie também um arquivo chamado user_serializer.py**.



No arquivo `user_serializer.py`, escreva a classe `UserSerializer`. Vamos escolher quais campos farão parte das operações da aplicação (farão parte do objeto JSON recebido e produzido e, por consequência, do objeto Python envolvido nas conversões).

Nota. A classe **Meta** existe a fim de manter o código organizado. Seu corpo contém, como seu nome sugere, metadados. São, em geral, dados de configurações. Classes de modelo também podem ter uma classe `Meta`. Veja esse exemplo.

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    class Meta:
        ordering = ['title']
```

Neste exemplo, a funcionalidade primária da classe `Book` é representar livros. A classe `Meta`, interna a ela, traz configurações. Neste exemplo, especifica que os campos devem ser ordenados de acordo com a coluna `title`.

Veja o código da serializadora de usuários.

```
from django.contrib.auth.models import User
from rest_framework import serializers

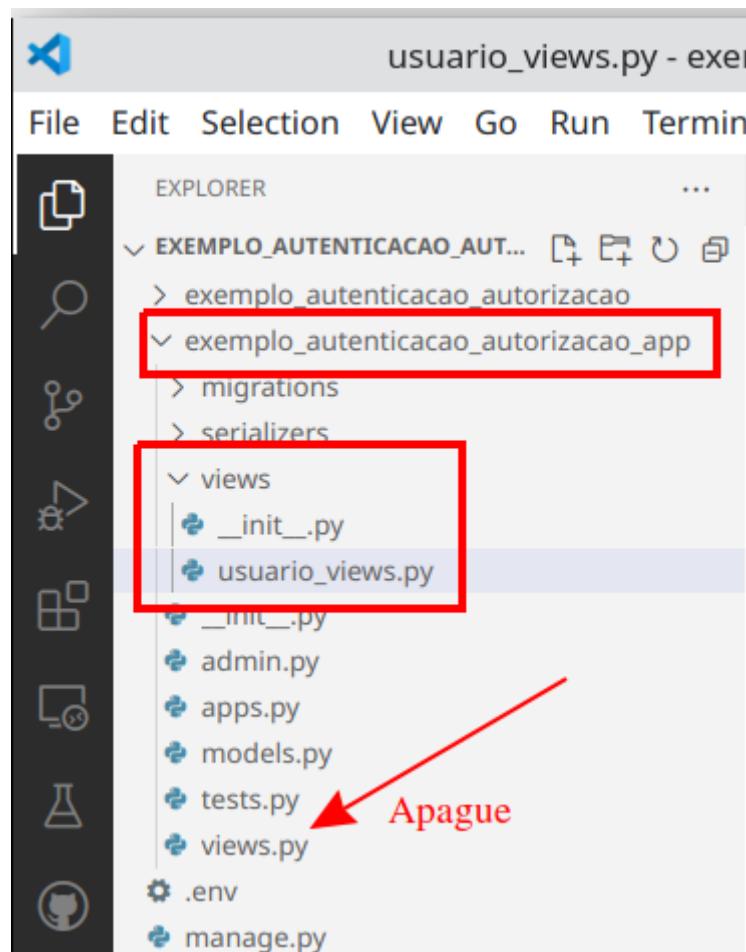
class UserSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True)

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'password')
```

A fim de simplificar o import a ser realizado em outros módulos externos, importe a classe `UserSerializer` no arquivo `__init__.py` que criamos há pouco.

```
#arquivo serializers/__init__.py
from .user_serializer import UserSerializer
```

A seguir, definimos uma View que viabiliza o cadastro de novos usuários. Para defini-la, crie uma pasta chamada `views` na raiz da aplicação. A seguir, crie um arquivo chamado `__init__.py` e outro chamado `usuario_views.py`.



Veja o código do arquivo **views/_init_.py**.

```
from .usuario_views import CadastroNovoUsuarioView
```

Veja as características da nova view.

- Se chama **CadastroNovoUsuarioView**.
- Herda de **views.APIView**. Essa é uma classe base para views do Django. Ela oferece acesso mais direto aos dados da requisição e nos permite especificar explicitamente os métodos do protocolo HTTP desejados.
- Possui um método chamado **post**. Ele é acionado quando uma requisição do tipo post for recebida. Como parâmetro, ele tem um objeto chamado `request`. Ele dá acesso aos dados recebidos na requisição.
- Instanciamos um `UserSerializer` e a ele entregamos os campos da `request`, acessíveis por meio do seu campo `data`. Internamente, **cada serializadora tem seu próprio campo também chamado `data`**. Esse é um **dicionário** que contém todos os dados sobre os quais a serializadora opera.

- Verificamos se os dados recebidos são “válidos” usando o método `is_valid`. Neste momento, por exemplo, a serializadora verifica, no banco de dados, se o `username` recebido já existe por lá. Se for o caso, a requisição terá como resposta um erro da família 400, ou seja, causado pela aplicação cliente. Há diversas validações padrão e também podemos especificar outras que nos sejam de interesse.
- Acessamos a coleção de usuários e criamos um novo usuário. Após a validação, os campos validados se encontram na propriedade `validated_data` da classe serializadora. Observe que **usamos o operador `**` para “desestruturar” o dicionário `validated_data`**. Isso acontece pois o **método `create_user` espera receber uma coleção de pares chave/valor “avulsos”, não incluídos em um dicionário**.

Nota. Inspire-se no código a seguir para lembrar sobre o funcionando do `kwargs` do Python.

```
def exibe_nomes(**nomes):
    for nome in nomes.values():
        print(nome)

#passando valores como argumentos nomeados, ou seja, chave=valor,
#avulsos, sem precisar de um dicionário
exibe_nomes(nome1='João', nome2='Maria', nome3='José')

#construindo um dicionário
pessoa = {'nome': 'João'}

#tentando passar o dicionário inteiro
#vai dar erro, pois, assim, o dicionário é um argumento posicional, e
#a função espera argumentos nomeados
exibe_nomes(pessoa)

#para passar um dicionário os dados do dicionário, como argumentos
#nomeados, usamos o operador **, desempacotando o dicionário
exibe_nomes(**pessoa)
```

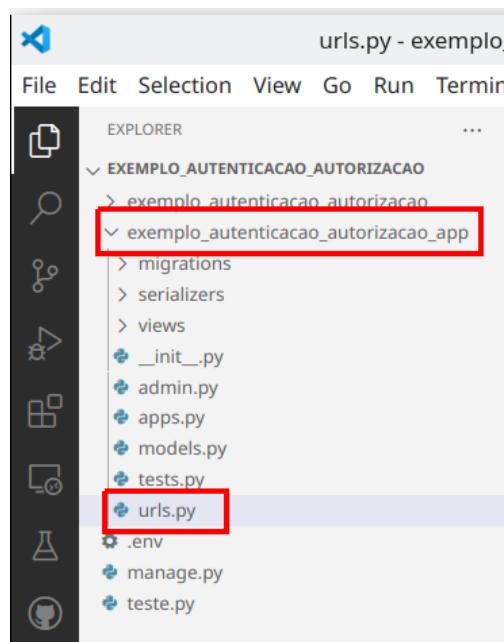
- Se o cadastro foi feito corretamente, devolvemos um objeto `Response`. Ele contém os dados de interesse (geralmente os dados do próprio objeto que foi criado) e o status do protocolo HTTP 201 Created, indicando que um recurso foi criado.
- Se o cadastro falhou, devolvemos o objeto `errors` da classe serializadora. Ele contém as mensagens de erro que explicam os erros eventualmente encontrados. Além disso, o código

de status do protocolo HTTP será 400 Bad Request, indicando que houve um erro causado pelo a requisição que foi enviada pelo cliente (e não foi um erro interno do servidor).

Veja o código.

```
from rest_framework import views, status
from rest_framework.response import Response
from exemplo_autenticacao_autorizacao_app.serializers import UserSerializer
from django.contrib.auth.models import User
class CadastroNovoUsuarioView(views.APIView):
    def post(self, request):
        serializer = UserSerializer(data=request.data)
        if serializer.is_valid():
            user = User.objects.create_user(**serializer.validated_data)
            return Response(UserSerializer(user).data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

A seguir, criamos um arquivo **urls.py** para definir os mapeamentos específicos da aplicação.



Veja seu conteúdo. Fazemos um primeiro mapeamento.

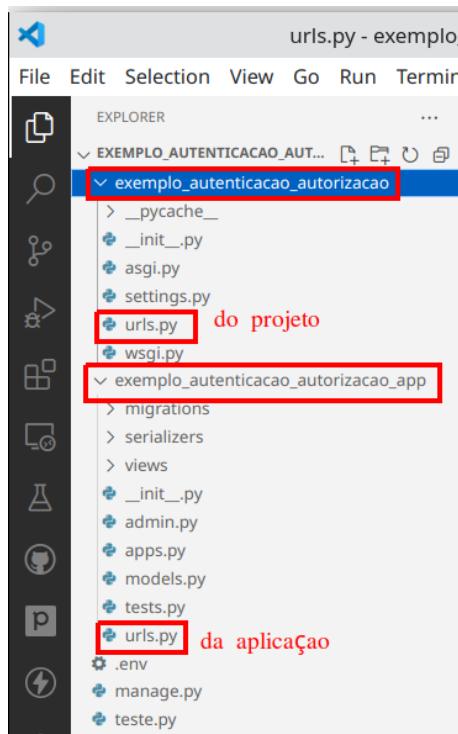
```
from django.urls import path
from exemplo_autenticacao_autorizacao_app.views import CadastroNovoUsuarioView
urlpatterns = [
    path('signup/', CadastroNovoUsuarioView.as_view(), name='signup'),
]
```

Nota. O parâmetro **name** serve para algumas coisas.

- **Reversão de URL:** Permite recuperarmos a URL, sem ser necessário digitá-la explicitamente no código, o que poderia ser pouco prático para manutenções futuras.
- **Uso da url em templates:** Num template (página HTML) você pode usar a URL por meio de seu nome, também sem ter de digitá-la explicitamente. Veja esse exemplo.

```
<a href="{% url 'signup' %}">Cadastre-se</a>
```

A seguir, precisamos incluir as urls da aplicação ao arquivo **urls.py do projeto**. Lembre-se que ele fica na raiz do projeto.

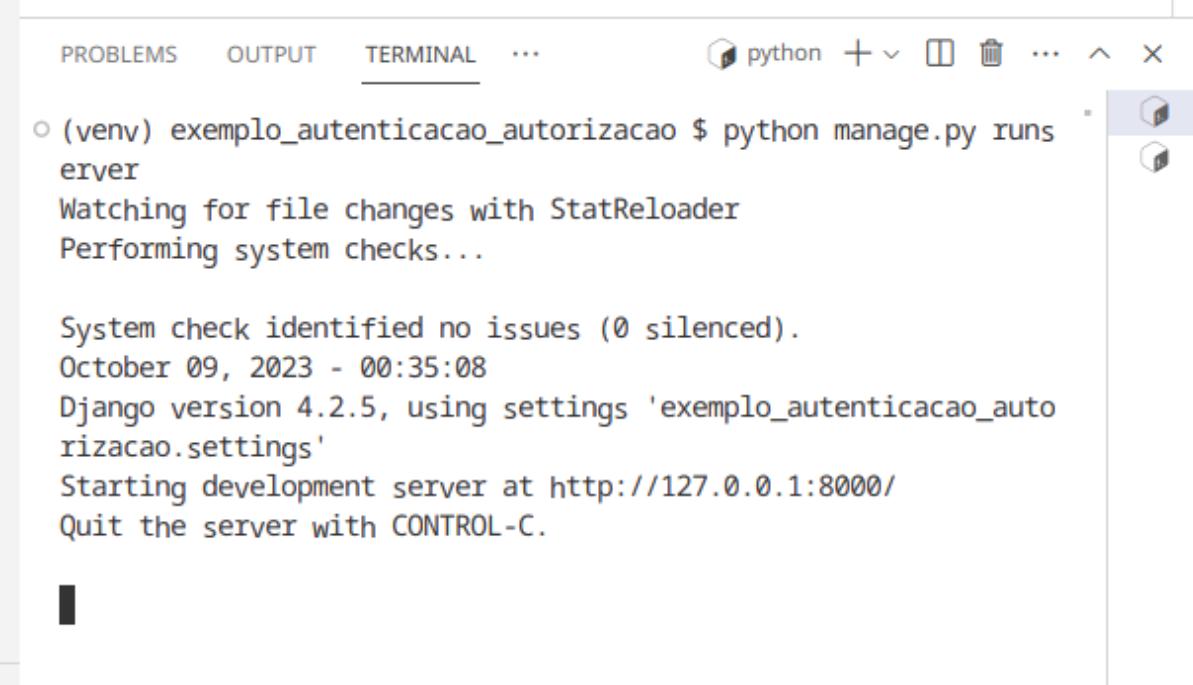


```
from django.contrib import admin
from django.urls import path, include

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

Certifique-se de que o servidor está em execução.

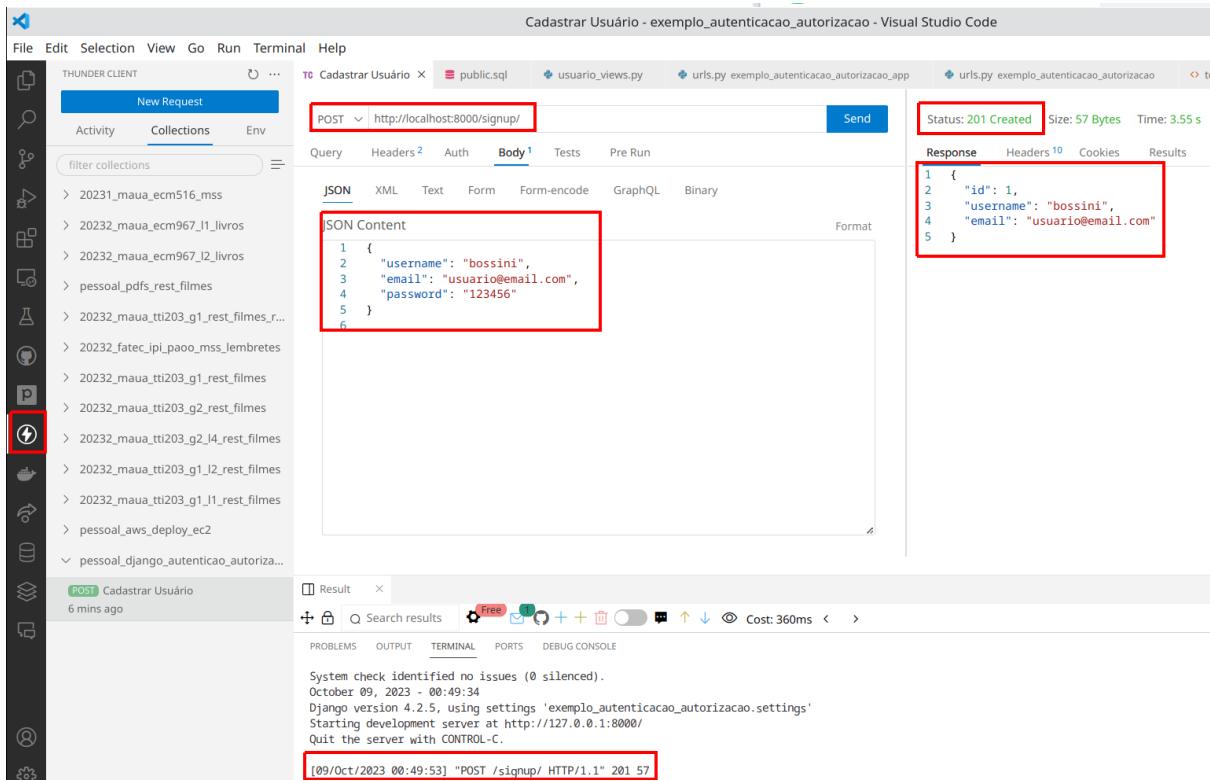
```
python manage.py runserver
```



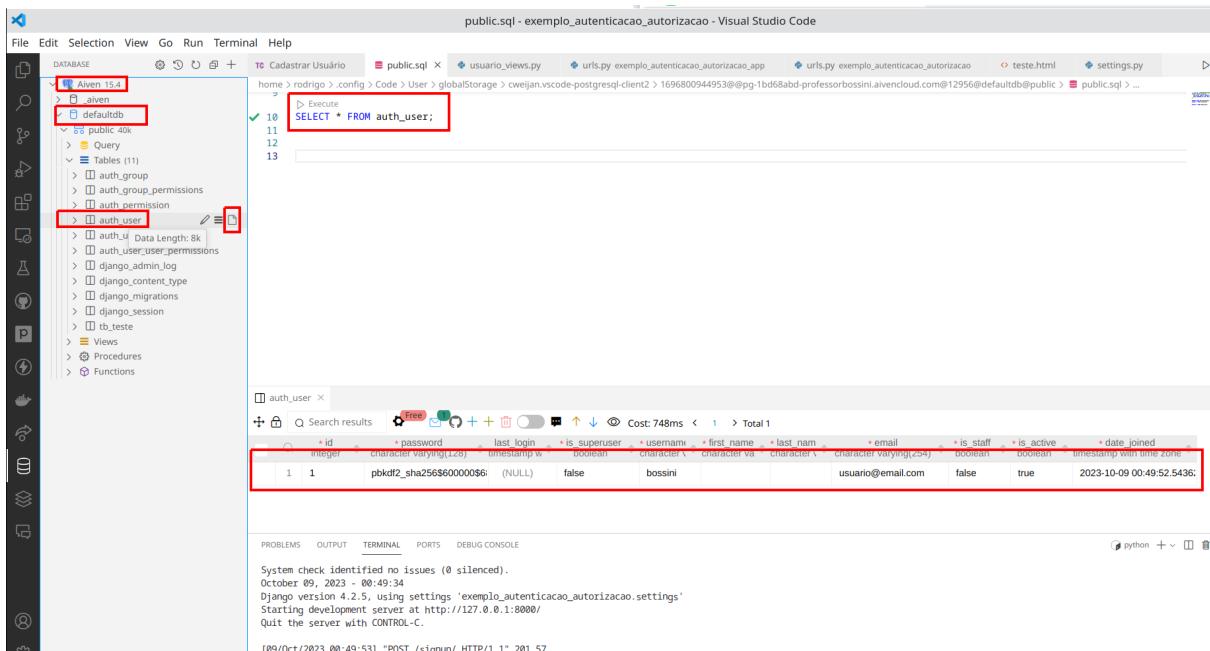
```
PROBLEMS OUTPUT TERMINAL ...
python + - x
○ (venv) exemplo_autenticacao_autorizacao $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
October 09, 2023 - 00:35:08
Django version 4.2.5, using settings 'exemplo_autenticacao_auto
rizacao.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Na Thunder Client, faça um teste.



Faça um SELECT no banco e verifique se o usuário foi, de fato, cadastrado.



Observe que a senha foi criptografada antes de ser armazenada. Inspecione também os demais campos.

De volta à Thunder Client, tente cadastrar um novo usuário com username igual ao do anterior. Veja o erro causado.

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

DATABASE

POST http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content

```

1 {
2   "username": "bossini",
3   "email": "usuario@email.com",
4   "password": "123456"
5 }
6

```

Response Headers 10 Cookies Results Docs

```

1 {
2   "username": [
3     "A user with that username already exists."
4   ]
5 }

```

auth_user

Search results Cost: 748ms Total 1

	id	password	last_login	is_superuser	username	first_name	last_name	email	is_staff	is_active	date_joined
1	1	pbkdf2_sha256\$600000\$6	(NULL)	false	bossini			usuario@email.com	false	true	2023-10-09 00:49:52.5436

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

Django version 4.2.5, using settings 'exemplo_autenticacao_autorizacao.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

[09/Oct/2023 00:49:53] "POST /signup/ HTTP/1.1" 201 57
Bad Request: /signup/
[09/Oct/2023 00:56:28] "POST /signup/ HTTP/1.1" 400 58

Altere o username. Ao mesmo tempo, especifique um e-mail inválido. Veja o erro.

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

DATABASE

POST http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content

```

1 {
2   "username": "bossini12",
3   "email": "nouser@email.com",
4   "password": "123456"
5 }
6

```

Response Headers 10 Cookies Results Docs

```

1 {
2   "email": [
3     "Enter a valid email address."
4   ]
5 }

```

auth_user

Search results Cost: 748ms Total 1

	id	password	last_login	is_superuser	username	first_name	last_name	email	is_staff	is_active	date_joined
1	1	pbkdf2_sha256\$600000\$6	(NULL)	false	bossini			usuario@email.com	false	true	2023-10-09 00:49:52.5436

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

[09/Oct/2023 00:49:53] "POST /signup/ HTTP/1.1" 201 57
Bad Request: /signup/
[09/Oct/2023 00:56:28] "POST /signup/ HTTP/1.1" 400 58
Bad Request: /signup/
[09/Oct/2023 00:58:15] "POST /signup/ HTTP/1.1" 400 99
Bad Request: /signup/
[09/Oct/2023 00:58:24] "POST /signup/ HTTP/1.1" 400 42

Observe que estamos admitindo senhas não muito seguras. Caso queiramos especificar uma validação específica para um determinado campo, temos diferentes alternativas.

- escrever um método **validate_nome_do_campo** na classe serializadora.
- sobrescrever o método **validate** da classe serializadora (isso nos permite validar diversos campos de uma só vez). Ele recebe um parâmetro “data” e pegamos um campo assim: `data.get('nome_do_campo')`
- escrever uma função num arquivo à parte, próprio para armazenar validadores. Depois numa classe de modelo, utilizá-lo. Veja um exemplo. **Apenas observe, não vamos utilizar este modo ainda.**

```
#arquivo qualquer
def validar_password(password):
    #logica de validacao
    #numa classe de modelo
    password = models.CharField(validators=[validar_password])
```

Neste momento, vamos utilizar a primeira estratégia. Na classe serializadora, escreva o seguinte método. Esta implementação informa que toda senha é inválida. Claro, vamos aprimorar a seguir. Estamos no arquivo **user_serializer.py**.

```
from django.contrib.auth.models import User
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True)

    #tem que ser esse nome: validate_nome_do_campo
    def validate_password(self, password):
        raise serializers.ValidationError("Senha inválida")

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'password')
```

Vamos validar utilizando **expressões regulares**. As regras serão as seguintes:

- Pelo menos uma letra maiúscula
- Pelo menos uma letra minúscula
- Pelo menos um número
- Pelo menos um símbolo especial
- Pelo menos oito caracteres

Se desejar, estude mais sobre expressões regulares em Python na página a seguir.

<https://docs.python.org/3/library/re.html>

Veja a implementação do validador.

Nota. Em Python, uma string escrita assim: `r'abc'` é uma '**raw string**'. Significa que seu conteúdo é tratado literalmente. Não há caracteres de escape antecedidos por barra invertida. É útil, por exemplo, para a especificação de diretórios.

```
#sem raw string
path1 = 'C:\\\\Users\\\\rodrigo\\\\Documents'

#com raw string
path2 = r'C:\\Users\\rodrigo\\Documents'
```

Também é útil no contexto de expressões regulares. Neste contexto, `\d` simboliza um dígito qualquer.

```
#sem raw string
digito_qualquer = '\\d'

#com raw string
digito_qualquer = r'\\d'
```

```

from django.contrib.auth.models import User
from rest_framework import serializers
import re

class UserSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True)

    #tem que ser esse nome: validate_nome_do_campo
    def validate_password(self, password):
        #pelo menos uma letra maiuscula
        if not re.search('[A-Z]', password):
            raise serializers.ValidationError("A senha deve conter pelo menos uma letra maiúscula")
        #pelo menos uma letra minuscula
        if not re.search('[a-z]', password):
            raise serializers.ValidationError("A senha deve conter pelo menos uma letra minúscula")
        #pelo menos um numero
        if not re.search('[0-9]', password):
            raise serializers.ValidationError("A senha deve conter pelo menos um número")
        #pelo menos um caracter especial (o ^ é para negar)
        if not re.search('[^a-zA-Z0-9]', password):
            raise serializers.ValidationError("A senha deve conter pelo menos um caracter especial")
        #pelo menos oito caracteres
        if len(password) < 8:
            raise serializers.ValidationError("A senha deve conter pelo menos oito caracteres")
        return password

    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'password')

```

No VS Code, faça novos testes. Veja alguns exemplos.

POST <http://localhost:8000/signup/>

Send

Response Headers 10 Cookies Results Docs

Status: 400 Bad Request Size: 68 Bytes Time: 2.85 s

1 {
2 "password": [
3 "A senha deve conter pelo menos uma letra maiúscula"
4]
5 }

POST <http://localhost:8000/signup/>

Send

Response Headers 10 Cookies Results Docs

Status: 400 Bad Request Size: 68 Bytes Time: 2.82 s

1 {
2 "password": [
3 "A senha deve conter pelo menos uma letra minúscula"
4]
5 }

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TC Cadastrar Usuário X public.sql usuario_views.py user_serializer.py urls.py exemplo_autenticacao_autorizacao_app urls.py exemplo_autenticacao_autorizacao teste.html setting

POST v http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "username": "{{#name}}",
3   "email": "{{#email}}",
4   "password": "FaltaUmDigito"
5 }
```

Status: 400 Bad Request Size: 58 Bytes Time: 2.81 s

Response Headers 10 Cookies Results Docs

```
1 {
2   "password": [
3     "A senha deve conter pelo menos um número"
4   ]
5 }
```

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TC Cadastrar Usuário X public.sql usuario_views.py user_serializer.py urls.py exemplo_autenticacao_autorizacao_app urls.py exemplo_autenticacao_autorizacao teste.html setting

POST v http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "username": "{{#name}}",
3   "email": "{{#email}}",
4   "password": "Cade0CaractereEspecial"
5 }
```

Status: 400 Bad Request Size: 68 Bytes Time: 2.80 s

Response Headers 10 Cookies Results Docs

```
1 {
2   "password": [
3     "A senha deve conter pelo menos um caracter especial"
4   ]
5 }
```

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TC Cadastrar Usuário X public.sql usuario_views.py user_serializer.py urls.py exemplo_autenticacao_autorizacao_app urls.py exemplo_autenticacao_autorizacao teste.html setting

POST v http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "username": "{{#name}}",
3   "email": "{{#email}}",
4   "password": "Curt01"
5 }
```

Status: 400 Bad Request Size: 63 Bytes Time: 2.80 s

Response Headers 10 Cookies Results Docs

```
1 {
2   "password": [
3     "A senha deve conter pelo menos oito caracteres"
4   ]
5 }
```

Cadastrar Usuário - exemplo_autenticacao_autorizacao - Visual Studio Code

File Edit Selection View Go Run Terminal Help

TC Cadastrar Usuário X public.sql usuario_views.py user_serializer.py urls.py exemplo_autenticacao_autorizacao_app urls.py exemplo_autenticacao_autorizacao teste.html setting

POST v http://localhost:8000/signup/ Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "username": "{{#name}}",
3   "email": "{{#email}}",
4   "password": "Tud0Certinho!"
5 }
```

Status: 201 Created Size: 79 Bytes Time: 3.12 s

Response Headers 10 Cookies Results Docs

```
1 {
2   "id": 2,
3   "username": "Erdman",
4   "email": "Buckridge.0e753f481bf@thunderclient.com"
5 }
```

2.7 (Usando a classe de modelo User do Django para fazer login) Para fazer login, o usuário precisa mostrar-se autêntico. Há diferentes mecanismos de autenticação, como Bearer Token Authentication, Basic, Digest entre outros. Neste material, vamos utilizar o mecanismo **Bearer Token Authentication com JWT (JSON Web Token)**.

Como funciona: Após o login bem-sucedido, o servidor envia um token que o cliente deve incluir nas solicitações subsequentes para acessar recursos protegidos. Esse token é geralmente incluído no cabeçalho da requisição, precedido pela palavra "Bearer".

Nota. Bearer significa algo como "detentor".

Em geral, quando utilizamos o mecanismo JWT (JSON Web Token), dois tipos de tokens estão envolvidos:

Access Token: O token de acesso (access token) é usado para acessar e autenticar requisições em endpoints protegidos. Ele tem todas as informações necessárias para identificar e autorizar um usuário.

Duração: Ele tem uma duração curta, o que significa que expira rapidamente. Isso é intencional, pois se um token de acesso for comprometido, ele poderá ser utilizado com más intenções por pouco tempo.

Informações: Normalmente, carrega informações sobre o usuário e/ou suas permissões. Isso permite que o servidor saiba quem está fazendo a requisição e o que essa pessoa tem permissão para acessar.

Refresh Token: O token de atualização (refresh token) não é usado para acessar endpoints diretamente. Ele é usado para obter um novo token de acesso quando o anterior expira.

Duração: Geralmente tem uma duração muito mais longa do que o token de acesso. Pode durar dias, semanas ou até mais, dependendo da implementação.

Segurança: Em muitas implementações, o refresh token é guardado de maneira segura no lado do servidor. Se um refresh token for comprometido, o impacto pode ser significativo, já que ele pode ser usado para obter tokens de acesso continuamente até sua expiração.

Rotação: Algumas implementações usam a técnica de rotação de token. Nela, cada vez que um refresh token é usado para obter um novo access token, um novo refresh token é também gerado e o anterior é invalidado. Isso garante que, se um refresh token for comprometido, ele só possa ser usado uma vez.

Para utilizar este mecanismo com Django, vamos usar o pacote **djangorestframework-simplejwt**.

Faça a sua instalação com

```
pip install djangorestframework-simplejwt
```

A seguir, no arquivo **settings.py**, vamos fazer os seguintes ajustes.

- Adicionar a nova aplicação à lista de aplicações instaladas no projeto.
- Apontar qual a classe responsável pela autenticação.
- Fazer configurações quanto ao funcionamento do mecanismo de autenticação JWT.

```

...
from pathlib import Path
import environ
#representa uma duração de tempo
from datetime import timedelta

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

SIMPLE_JWT = {
    #quanto tempo o token vai durar
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    #quanto tempo o refresh token vai durar
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    #se o refresh token vai ser rotacionado
    'ROTATE_REFRESH_TOKENS': False,
    #algoritmo de criptografia
    'ALGORITHM': 'HS256',
    #chave de criptografia
    'SIGNING_KEY': env('SECRET_KEY'),
    #tipos de autenticação, vamos usar apenas o Bearer. Tem que ser uma
    #tupla, por isso a vírgula no final
    'AUTH_HEADER_TYPES': ('Bearer',),
}
...

```

O próximo passo é dar acesso às funcionalidades de obtenção dos tokens de acesso e de atualização. Para tal, podemos usar views definidas pelo próprio `rest_framework_simplejwt` que instalamos. No arquivo **urls.py do projeto**, faça o ajuste a seguir.

```
from django.contrib import admin
from django.urls import path, include
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView

urlpatterns = [
    path("admin/", admin.site.urls),
    #para obter um access token e um refresh token em função de um username e password
    path('token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    #para obter um novo access token em função de um refresh token
    path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('', include('exemplo_autenticacao_autorizacao_app.urls'))
]
```

Reinic peace a aplicação com

```
python manage.py runserver
```

Observe que a aplicação que instalamos (`token_blacklist`) possui migrações necessárias para seu correto funcionamento.



```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

● (venv) exemplo_autenticacao_autorizacao $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 11 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): token_blacklist.
Run 'python manage.py migrate' to apply them.
October 09, 2023 - 03:00:49
Django version 4.2.5, using settings 'exemplo_autenticacao_autorizacao.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

○ ^C(venv) exemplo_autenticacao_autorizacao $
```

Por isso, execute

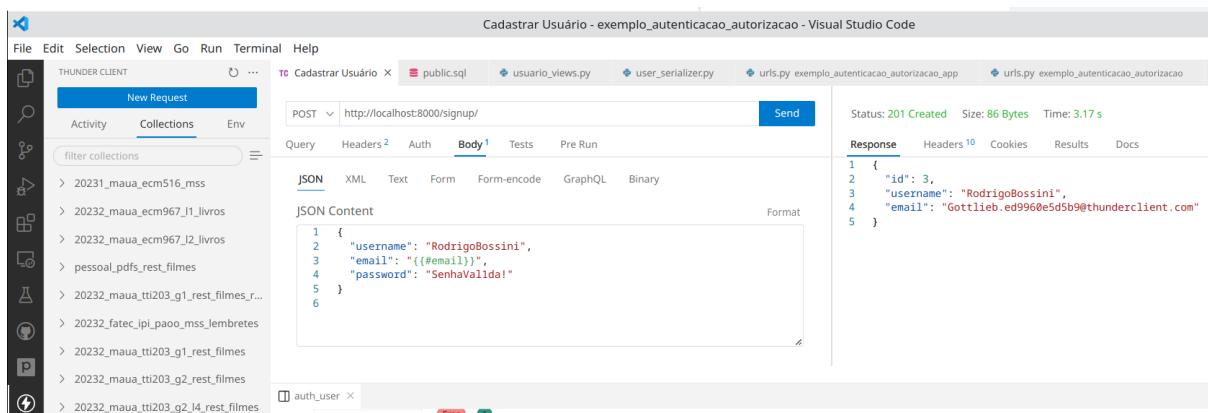
```
python manage.py migrate
```

```
PyCharm | Community Edition 2022.3.1
● (venv) exemplo_autenticacao_autorizacao $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, token_blacklist
Running migrations:
  Applying token_blacklist.0001_initial... OK
  Applying token_blacklist.0002_outstandingtoken_jti_hex... OK
  Applying token_blacklist.0003_auto_20171017_2007... OK
  Applying token_blacklist.0004_auto_20171017_2013... OK
  Applying token_blacklist.0005_remove_outstandingtoken_jti... OK
  Applying token_blacklist.0006_auto_20171017_2113... OK
  Applying token_blacklist.0007_auto_20171017_2214... OK
  Applying token_blacklist.0008_migrate_to_bigautofield... OK
  Applying token_blacklist.0010_fix_migrate_to_bigautofield... OK
  Applying token_blacklist.0011_linearizes_history... OK
  Applying token_blacklist.0012_alter_outstandingtoken_user... OK
○ (venv) exemplo_autenticacao_autorizacao $
```

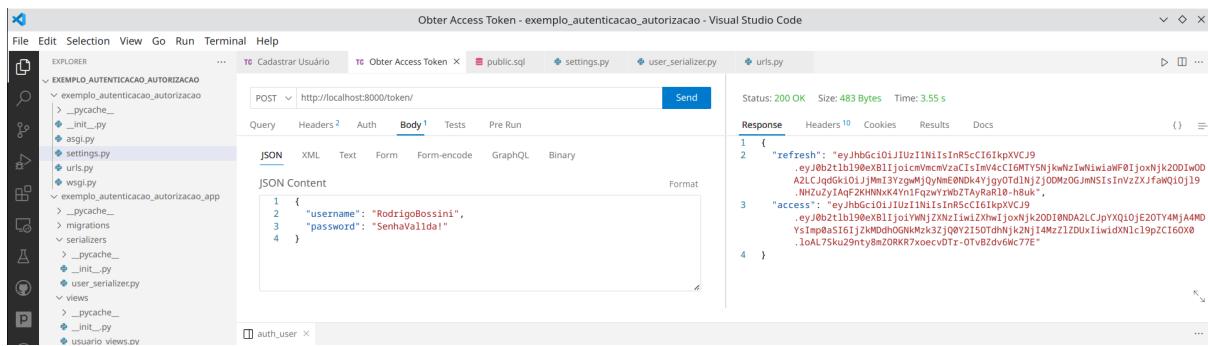
Coloque o servidor em execução uma vez mais com

```
python manage.py runserver
```

Para testar, comece criando um usuário com a Thunder Client, caso ainda não tenha um ou não se lembre da senha daqueles que já foram cadastrados.



Agora tente obter um token de acesso. Observe que o método a ser usado é o POST.



A resposta contém um access token e um refresh token. Copie o refresh token e faça novo teste, a fim de obter um novo access token.

