A partir de agora criaremos um projeto chamado “**smart\_city**”.

Este projeto vai comportar uma aplicação com a finalidade principal de coletar e expor os dados dos sensores da cidade inteligente. Para isso construiremos algumas APIs juntamento com o Banco de Dados da aplicação.

# 1 – Criando o projeto “smart\_city”

1. Crie uma pasta chamada **smart\_city** pelo qual hospedará esse projeto.
2. Crie um **ambiente virtual**.

**py -m venv env**

1. Ative esse ambiente.

**env\scripts\activate**

1. Instale o Django.

**pip install Django**

1. Faça a instalação do Django Rest Framework executando o seguinte comando no terminal:

**pip install djangorestframework**

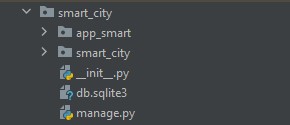
1. Feitos os passos acima, vamos criar o projeto **smart\_city** no Django com o comando:

**django-admin startproject smart\_city .**

1. Tendo criado o projeto com sucesso, acessamos a pasta do projeto e criamos o **app\_smart** com o comando

**py manage.py startapp app\_smart**

1. Feito esses passos a estrutura de pastas deve ficar como segue:



1. Acesse o arquivo **settings.py** que está dentro da pasta smart\_city do projeto e na sessão INSTALLED\_APPS acrescente as seguintes linhas:

Texto

Descrição gerada automaticamente

1. Para que seja criado a estrutura de tabelas que o Django irá utilizar no projeto executamos o comando **migrate** como segue:

**py manage.py migrate**

Confira com o **db.sqlite3** se as tabelas do Django foram criadas.

1. Com as tabelas criadas já é possível criar um *superusuario* para o nosso *app\_smart* executando o comando que segue:

**py manage.py createsuperuser**

1. Para facilitar nosso trabalho mais a frente utilize o seguinte nome de usuário e senha para o superuser:

**username = ‘lin’**

**password = ‘123’**

Crie a rota de inicialização da aplicação editando o arquivo **urls.py** do projeto.

Texto

Descrição gerada automaticamente

1. Criar o arquivo **urls.py** do **app\_smart**:

Texto

Descrição gerada automaticamente

1. Como criamos no arquivo acima a chamada da função ‘*abre\_index*’ vamos criar essa função em **views.py:**

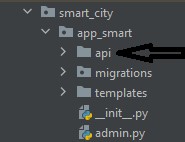
Texto

Descrição gerada automaticamente

# 2 – Criando a API para cadastro de novos usuários

Dois arquivos são chaves no desenvolvimento de APIs no Django. São eles: **serializers.py** e **viewsets.py.**

Como boa prática em um sistema que tenha várias funções sendo acessadas por *templates* e por *APIs*, costumamos colocar os arquivos referentes a APIs em uma pasta específica chamada **api.** Crie essa pasta dentro da pasta do seu **app\_smart** ficando como segue:



## 2.1 – Criando o arquivo serializers.py

Dentro da pasta **api** crie um arquivo chamado **serializers.py e** implemente o seguinte código:

Texto

Descrição gerada automaticamente

Serializar significa definir uma classe que irá converter objetos Python em representações que possam ser facilmente renderizadas em formatos específicos como JSON ou XML por exemplo, e vice-versa.

Um serializador pode realizar várias tarefas como:

1. ***Validação de Dados***: Ele valida os dados de entrada para garantir que estejam corretos antes de serem salvos no banco de dados.
2. ***Conversão de Dados***: Ele converte objetos Python em formatos de dados que podem ser enviados pela web, como JSON.
3. ***Deserialização de Dados***: Ele desserializa os dados recebidos do cliente de volta para objetos Python para processamento.
4. ***Definição de Campos***: Ele determina quais campos do modelo devem ser incluídos ou excluídos na resposta da API.
5. ***Controle de Acesso****:* Ele controla quais campos podem ser lidos ou gravados pelos usuários da API.

A linha **extra\_kwargs = {'password': {'write\_only': True}}** é usada para especificar argumentos extras para o campo *password* no serializador.

Aqui no Django REST Framework, é usado para configurar opções adicionais como um campo deve ser serializado ou desserializado. No nosso caso, **write\_only: True** significa que o campo *password* só será usado durante a desserialização, ou seja, quando você está criando ou atualizando um usuário e não será incluído na serialização, ou seja, quando é enviado como resposta a uma solicitação.

Isso é útil para garantir que a senha não seja incluída nas respostas da API, o que não seria uma prática segura.

## 2.2 - Criando o arquivo viewsets.py

Também dentro da pasta **api** crie um arquivo chamado **viewsets.py** eimplemente o seguinte código:

Texto

Descrição gerada automaticamente

Os atributos ***queryset***, ***serializer\_class*** e ***permission\_classes*** são nomes convencionais e esperados pelo Django REST Framework. Alterá-los causará erros, pois o DRF espera encontrar esses atributos com esses nomes específicos para configurar corretamente a ViewSet.

Logo em seguidacria o método **post** o qual delega o processamento da requisição para o método **create** com **self.create(request, \*args, \*\*kwargs)** fazendo com que seja criado uma instância da classe do serializador que no nosso caso é uma instância de **User.**

## 2.3 – Criando a rota para a API

No arquivo **urls.py** do **app\_smart** acrescente as seguintes linhas:

Texto

Descrição gerada automaticamente

Isso fará com que nossa API já esteja disponível.

## 2.4 – Testando a API

**2.4.1 – Testando a API via admin do Django** Execute o servidor com o comando:

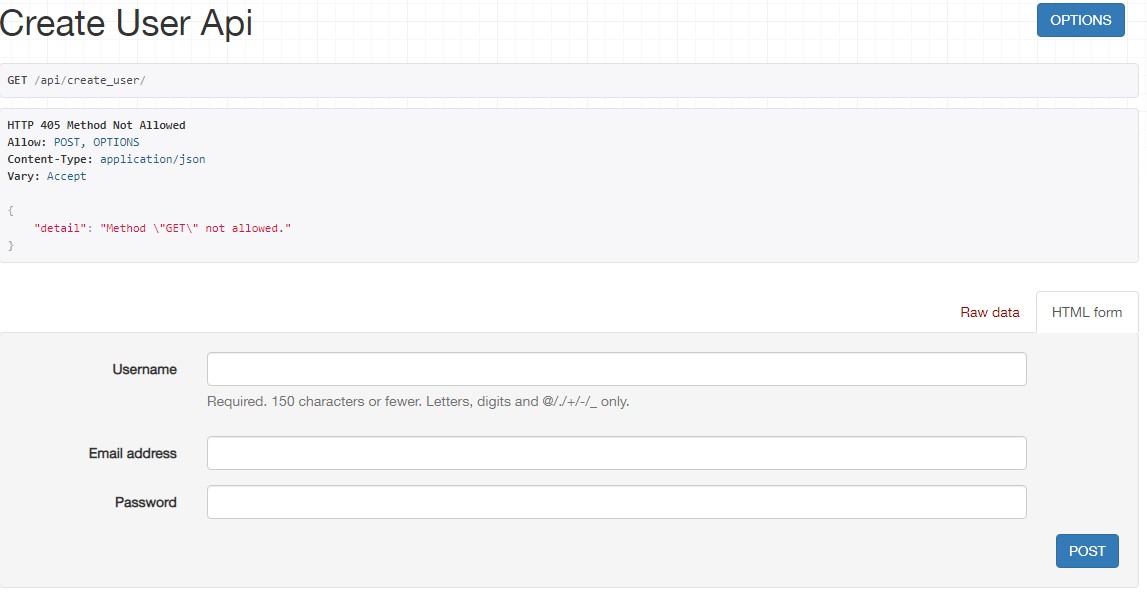
**py manage.py runserver**

No Browser acesso o endereço da API **http://127.0.0.1:8000/api/create\_user/**



Se a página anterior foi exibida, sinal de que sua API já está funcionando. Porém ela apresenta uma mensagem de que as credenciais não foram fornecidas. Isto acontece porque em **viewsets.py** foi criado a seguinte linha **permission\_classes = [IsAdminUser]**. Isto **pede** que as credenciais de um ***superuser*** sejam enviadas no cabeçalho da requisição POST da API.

Comente essa linha no código (colocando # antes da linha) e execute novamente o servidor. A página que segue deve ser exibida já permitindo que você faça o cadastro de um novo usuário

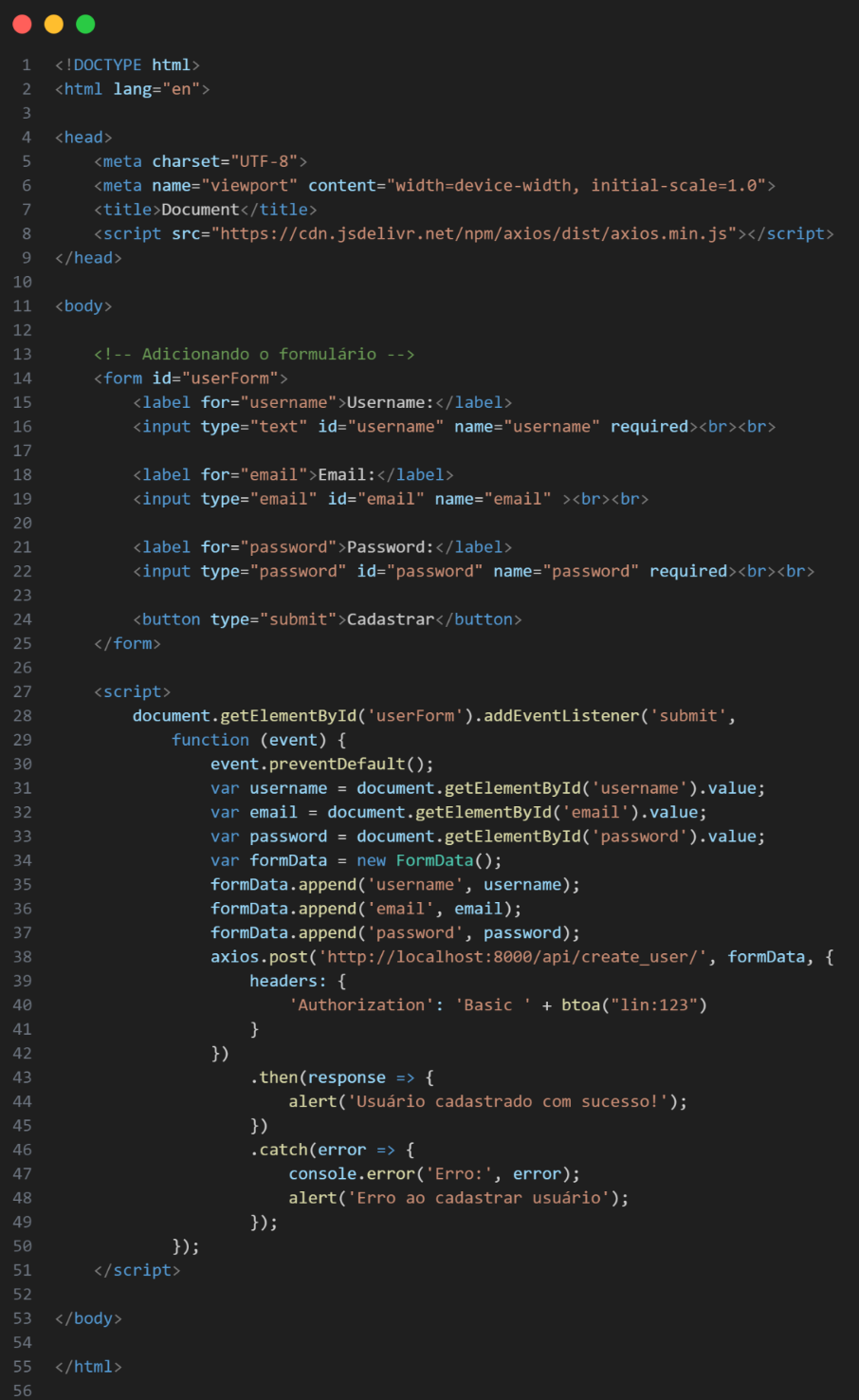
.

Retire o comentário da linha **permission\_classes = [IsAdminUser]** para que a API volte a solicitar as credenciais de um *superuser* para dar acesso a API.

### 2.4.2 – Testando a API via página html com os comandos *fetch* e *axios*

Para acessar a API a partir de uma aplicação Front End enviando os parâmtros da credencial de um super usuarios, você pode utilizar os comandos “*Fetch*” e “*Axios*”. A seguir temos um exemplo de cada enviando como credenciais o *superusuario* criado quando da criação do projeto que foi **username=*lin***  e **password=*123***

Exemplo **Axios**:

****

## <!DOCTYPE html>

## <html lang="en">

## <head>

## <meta charset="UTF-8">

## <meta name="viewport" content="width=device-width, initial-scale=1.0">

## <title>Document</title>

## <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

## </head>

## <body>

## <!-- Adicionando o formulário -->

## <form id="userForm">

## <label for="username">Username:</label>

## <input type="text" id="username" name="username" required><br><br>

## 

## <label for="email">Email:</label>

## <input type="email" id="email" name="email" ><br><br>

## 

## <label for="password">Password:</label>

## <input type="password" id="password" name="password" required><br><br>

## 

## <button type="submit">Cadastrar</button>

## </form>

## <script>

## document.getElementById('userForm').addEventListener('submit',

## function (event) {

## event.preventDefault();

## var username = document.getElementById('username').value;

## var email = document.getElementById('email').value;

## var password = document.getElementById('password').value;

## var formData = new FormData();

## formData.append('username', username);

## formData.append('email', email);

## formData.append('password', password);

## axios.post('http://localhost:8000/api/create\_user/', formData, {

## headers: {

## 'Authorization': 'Basic ' + btoa("lin:123")

## }

## })

## .then(response => {

## alert('Usuário cadastrado com sucesso!');

## })

## .catch(error => {

## console.error('Erro:', error);

## alert('Erro ao cadastrar usuário');

## });

## });

## </script>

## 

## </body>

## </html>

## 

## 2.4 – Fazendo com que o ‘password’ seja armazenado criptografado

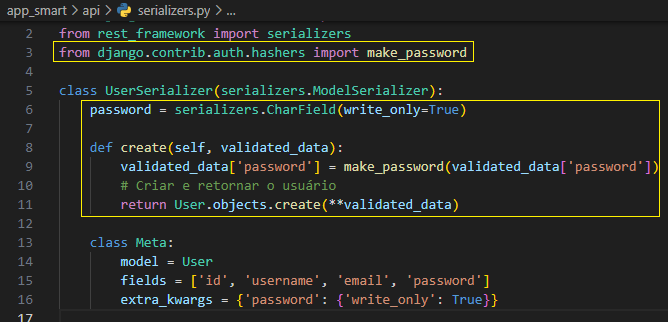
Após ter criado um novo usuário, abre o db.sqlite3 e verifique os dados do usuário que você cadastrou. No campo *‘password’* observe que foi armazenado sem criptografia.

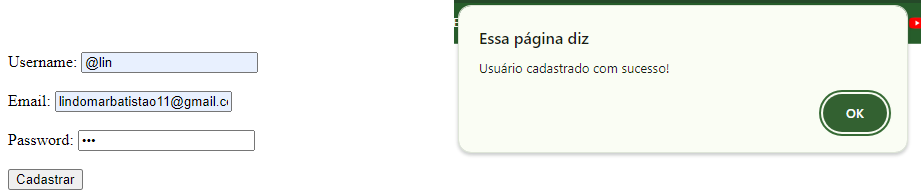
Linha do tempo

Descrição gerada automaticamente com confiança baixa

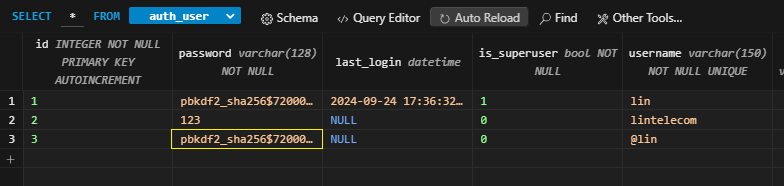
Vamos corrigir isso implementando a criptografia para este campo.

Abre o arquivo *serializers.py* e acrescente o seguinte código:





Execute novamente a API e faça o cadastro de um novo usuário. Depois verifique no BD como foi armazenado a senha.

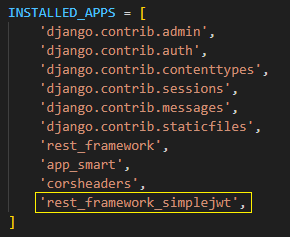


# 3 – Instalando a autenticação JWT na API

## 3.1 – Instalando e configurando as bibliotecas JWT(Jason Web Token)

**pip install djangorestframework-simplejwt**

Agora acesse o arquivo **settings.py** que está dentro da pasta *smart\_city* do projeto e na sessão INSTALLED\_APPS acrescente as seguintes linhas:



Acrescente a configuração abaixo no arquivo *settings.py*:

Texto

Descrição gerada automaticamente

REST\_FRAMEWORK = {

    'DEFAULT\_AUTHENTICATION\_CLASSES': (

        'rest\_framework\_simplejwt.authentication.JWTAuthentication',

        'rest\_framework.authentication.SessionAuthentication',

    )

}

Agora que utilizaremos a autenticação JWT o que fará com que um token seja solicitado para que você consiga acessar o API, acrescentaremos a sessão a seguir para configurar o tempo de validade do token. Acrescente então a seguinte sessão:

Texto

Descrição gerada automaticamente

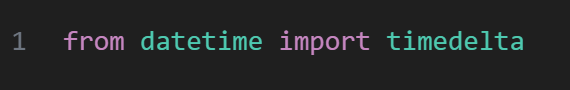
SIMPLE\_JWT = {

    'ACCESS\_TOKEN\_LIFETIME': timedelta(minutes=15),

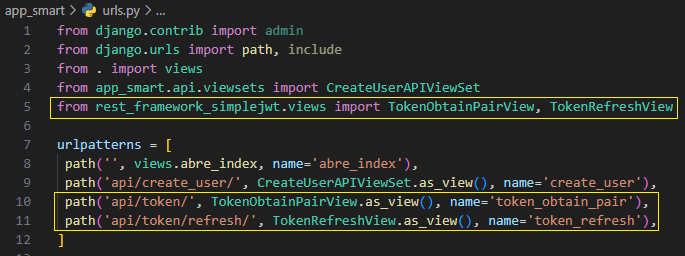
    'REFRESH\_TOKEN\_LIFETIME': timedelta(days=7),

}

Isso exige que você faça a importação da biblioteca *timedelta.* Acrescente a seguinte linha no começo do arquivo *settings.py*



Crie as URLs para obtenção dos Tokens de acesso, abra o arquivo *urls.py* do seu***app\_smart***e acrescente as seguintes linhas:



Após a instalação do JWT é necessário executar o comando **migrate** para que a tabela necessária para aramazenar os tokens JWT seja criada.

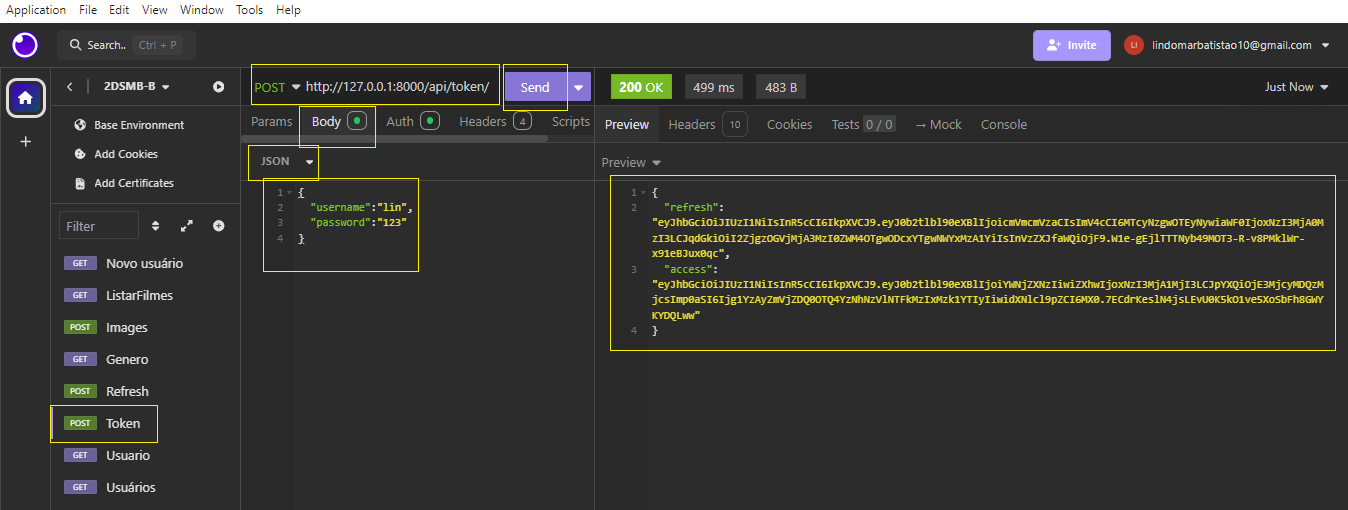
**python manage.py migrate**

## 3.2 – Testando o acesso à API com Token

**3.2.1 – Testando API utilizando o software Insomnia** Execute o servidor da API:

**py manage.py runserver**

Abre o Insomnia e crie um HTTPRequest conforme imagem:



1. – Clique em “+” e crie em novo HTTPRequest – No exemplo criei *Obter Token smart\_city*
2. *–* Informe a URL criada para obtenção do token ‘[http://127.0.0.1:8000/api/token/](http://127.0.0.1:8000/api/token) e selecione a opção ‘POST’
3. – Selecione a opção *Body 🡪 JSON*
4. – Monte um arquivo JSON com username e password conforme imagem. Utilize o *username=’lin’* e o *password=’123’*
5. *{*

*"username": "lin",*

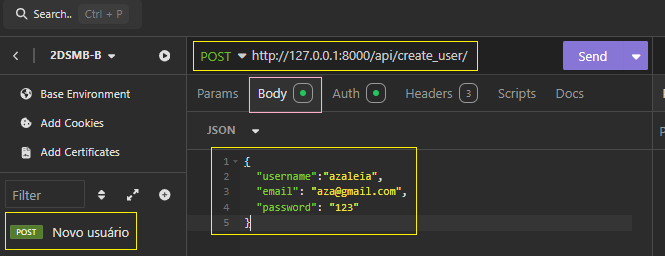
*"password": "123"*

*}*

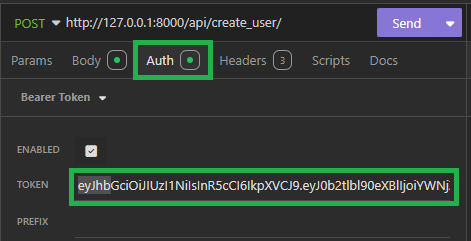
1. – Clique em *‘Send’.*
2. *– Copie o ‘Token’ gerado.*

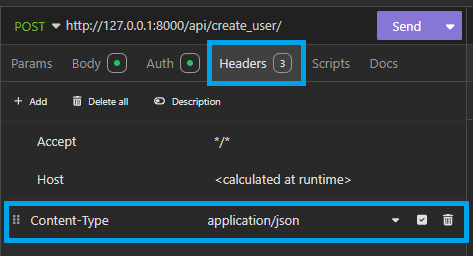
*-------------------------------------------------------------------------------------------*

Agora acessaremos a API para criar um usuário informando o token obtido no cabeçalho (Header) da requisição. Seguimos os passos conforme a imagem:



No 5o. passo você deve informar o *token* obtido como mostrado na figura que segue:





Feito isso clique no botão “**Send**” e estando tudo certo deve retornar o código ‘**201 Created**’ conforme imagem que segue:

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

# 4 – Criando uma API para cadastro de sensores

Para a criação de uma API devemos seguir os passos:

* *definir o model*
* *criar serializers*
* *criar viewsets*
* *criar as rotas de acesso*

## 4.1 – Criando o model Sensor

Abra o arquivo *models.py* do *app\_smart* e edite como segue:

Texto

Descrição gerada automaticamente

from django.db import models

class Sensor(models.Model):

    TIPOS\_SENSOR\_CHOICES = [

        ('Temperatura', 'Temperatura'),

        ('Umidade', 'Umidade'),

        ('Contador', 'Contador'),

        ('Luminosidade', 'Luminosidade'),

    ]

    tipo = models.CharField(max\_length=50, choices=TIPOS\_SENSOR\_CHOICES)

    mac\_address = models.CharField(max\_length=20, null=True)

    latitude = models.FloatField()

    longitude = models.FloatField()

    localizacao = models.CharField(max\_length=100)

    responsavel = models.CharField(max\_length=100)

    status\_operacional = models.BooleanField(default=True)

    observacao = models.TextField(blank=True)

    def \_\_str\_\_(self):

        return f"{self.tipo} - {self.localizacao}"

Em TIPOS\_SENSOR\_CHOICES  nós especificamos quais os “Tipos de Sensores” suportados. Como estes tipos não são uma lista infinita, especificamos os tipos nesta lista. Se futuramente precisarmos acrescentar mais algum tipo basta adicionar na lista o novo tipo.

O campo **mac\_address** armazenará o endereço físico do sensor (da placa ESP).

Depois de criar o código do arquivo *models.py* execute o comando *makemigrations* e em seguida o comando *migrate*.

**py manage.py makemigrations**

Algumas linhas serão exibidas no terminal.

**py manage.py migrate.**

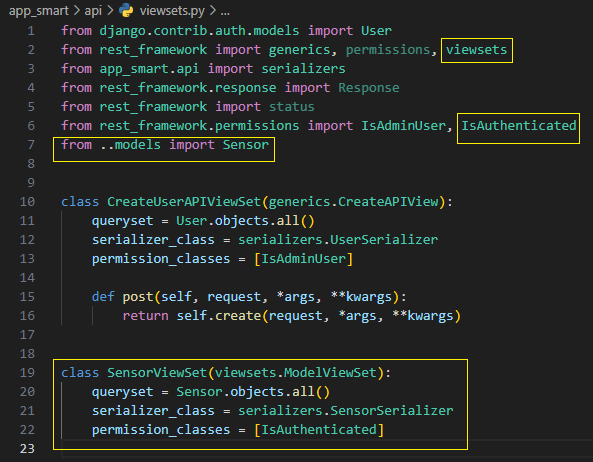
## 4.2 – Incluindo o model Sensor no arquivo serializers.py

Abra o mesmo arquivo ***serializers.p*y** na pasta **api** em **smart\_city.**

Texto

Descrição gerada automaticamente

## 4.3 – Incluindo a classe Sensor no arquivo viewsets.py



## 4.4 – Criando rotas de acesso para a API

Abra o arquivo ***urls.py*** do ***app\_smart.***

Acrescente as importações

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

De modo geral, quando criamos uma API temos que criar uma rota para responder a cada método de requisição para a API, ou seja, teria que ter uma rota para GET, outra para POST e assim por diante.

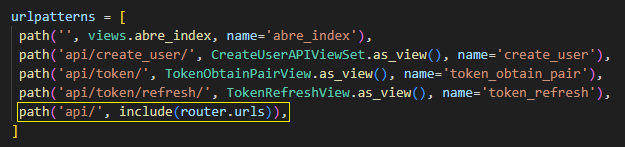
Utilizar a biblioteca ***DefaultRouter*** do *Django* otimiza significativamente esse processo, permitindo que, com apenas uma URL seja possível acomodar todos os métodos de requisição HTTP(GET, POST, PUT, DELETE). O ***DefaultRouter*** gerencia automaticamente as rotas paraos diferentes métodos HTTP com base nas ações definidas em ViewSets, simplificando a configuração das URLs da API.

Depois das importações acrescente as seguinte linhas:



A letra **‘*r*’** antes de ‘*sensores*’ serve para indicar que se trata de uma ‘*string bruta’.* Caso a string venha acompanhada de ‘\’ (barra invertida), essa barra não será considerada como caractere ‘scape’ do Python, como por exemplo ‘\n’ que criaria uma nova linha.

E na sessão urlpatterns acrescente a seguinte linha:



Com isso a nossa rota para a API que acabamos de criar vai ser [***http://127.0.0.1:8000/api/sensores***](http://127.0.0.1:8000/api/sensores)

**4.5 – Testando API utilizando o software Insomnia** Execute o servidor da API:

**py manage.py runserver**

Execute o mesmo procedimento descrito no tópico **3.2.1** para obter o ***token*** de acesso.

Agora crie um novo HTTPRequest conforme figura que segue:



1. – Clique em “+” e crie em novo HTTPRequest – No exemplo criei *“Cadastrando Sensor Novo”*
2. *–* Informe a URL criada para obtenção do token

‘ [http://127.0.0.1:8000/api/ sensores/](http://127.0.0.1:8000/api/token) e selecione a opção ‘POST’

1. – Selecione a opção *‘JSON’*
2. – Monte um arquivo JSON com os dados do sensor a ser cadastrado conforme segue:

{

"tipo": "Umidade",

"latitude": -23.345255,

"longitude": -47.12338,

"localizacao": "LAB A105",

"responsavel": "Aluno - FULANO",

"unidade\_medida": "oC.",

"status\_operacional": true,

"observacao": "Sensor cadastrado pela API no Insomnia"

}

Texto

Descrição gerada automaticamente

Agora informe o *token* obtido na aba ***Auth*** conforme a imagem que segue:

Tela de celular com aplicativo aberto

Descrição gerada automaticamente

1. – No campo ao lado digite ‘**Bearer**’ seguido do *token* obtido

Feito isso clique no botão “**Send**” e estando tudo certo deve retornar o código ‘**201 Created**’ conforme imagem que segue:

Texto

Descrição gerada automaticamente

# 5 – Aplicando filtros na API sensores

Em muitas situações é necessário que a consulta seja realizada com filtros como exemplo podemos desejar fazer uma consulta à API *sensores* somente dos registros onde “*status operacional=True*”. Ou ainda podemos desejar selecionar somente registros onde o “Tipo” seja “Temperatura” ou ainda onde o campo “localizacao=A103”, ou ainda podemos desejar selecionar somente os registros onde o campo “responsavel=joao”.

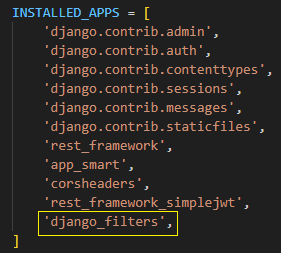
Portanto, vamos implementar os filtros na nossa API sensores.

## 5.1 – Instalando e configurando django-filter

Acesse o terminal e execute o comando como segue:

**pip install django-filter**

Agora acesse o arquivo **settings.py** que está dentro da pasta *smart\_city* do projeto e na sessão INSTALLED\_APPS acrescente a seguinte linha:



## 5.2 – Criando o arquivo filters.py

Na pasta **api** do **app\_smart**, crie um arquivo com o nome **filters.py**

Digite neste arquivo o código abaixo:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

import django\_filters

from app\_smart.models import Sensor

class SensorFilter(django\_filters.FilterSet):

    responsavel = django\_filters.CharFilter(field\_name='responsavel', lookup\_expr='icontains')

    status\_operacional = django\_filters.CharFilter(field\_name='status\_operacional', lookup\_expr='exact')

    tipo = django\_filters.CharFilter(field\_name='tipo', lookup\_expr='exact')

    localizacao = django\_filters.CharFilter(field\_name='localizacao', lookup\_expr='icontains')

    class Meta:

        model = Sensor

        fields = ['status\_operacional', 'tipo', 'localizacao', 'responsavel']

Observe que neste arquivo especificamos quais os campos que desejamos que funcionem como busca na API Sensores. Neste caso especificamos os campos **‘*status\_operacional’, ‘tipo’, ‘localizacao’*** *e* ***‘responsavel’****.*

Para os campos ‘***responsavel*’** e ‘***localizacao*’** usamos o argumento **lookup\_expr='icontains'**, isso significa que a busca pode se dar por qualquer parte do dado cadastrado no campo esteja ela cadastrado em maiúsculo ou minúsculo.

Por exemplo: Supondo existam 2 registros contendo os nomes: ‘J*oão da Silva*’ e outro como o nome ‘Rafael e Silva’ cadastrados no campo *‘responsavel’*.

Se na busca eu informar apenas ‘*silva*’ os dois registros serão encontrados, ou seja, a expressão **lookup\_expr='icontains'** é equivalente a expressão “**like**” em SQL.

Já para os campos ‘*tipo*’ e ‘*status\_operacional*’ usamos o argumento **lookup\_expr='exact'** o que significa que a busca tem que ser feita exatamente como cadastrada.

**5.3 – Configurando o arquivo viewsets.py para usar filtros na API sensores pelo método GET.**

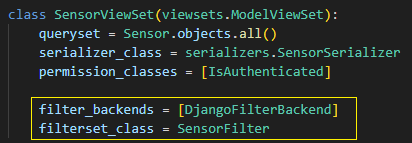
Abra o arquivo *viewsets.py*

Acrescente as importações:

Tela preta com letras brancas

Descrição gerada automaticamente

E na classe *SensorViewSet* acrescente as seguintes linhas:

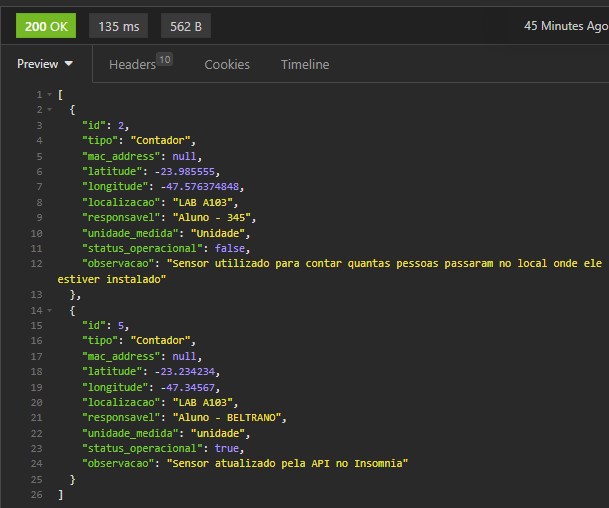


## 5.4 – Testando API sensores com filtros pelo software Insomnia

Agora crie consultas no Insomnia para os filtros como parâmetros na URL como segue o exemplo:

# http://127.0.0.1:8000/api/sensores/?tipo=Contador&localizacao=a1

Esta consulta retorna todos os sensores onde o *tipo* é exatamente “Contador” e a *localizacao* contenha “a1”. Um resultado exemplo é o que segue:

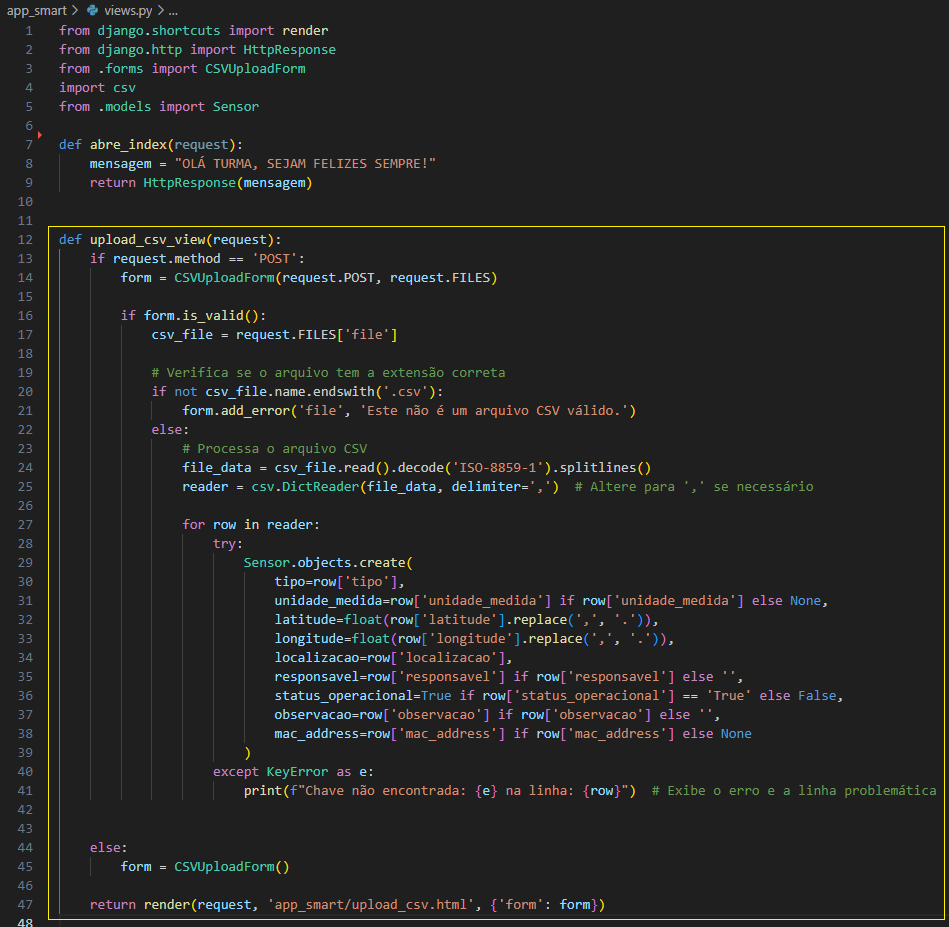


# http://127.0.0.1:8000/api/sensores/?tipo=Umidade&localizacao=LAB A105



# 6 – Fazendo uma carga de dados de sensores

Para popular o banco de dados com arquivo csv crie o método upload\_csv\_view conforme figura abaixo:



from django.shortcuts import render

from django.http import HttpResponse

from .forms import CSVUploadForm

import csv

from .models import Sensor

def abre\_index(request):

mensagem = "OLÁ TURMA, SEJAM FELIZES SEMPRE!"

return HttpResponse(mensagem)

def upload\_csv\_view(request):

if request.method == 'POST':

form = CSVUploadForm(request.POST, request.FILES)

if form.is\_valid():

csv\_file = request.FILES['file']

# Verifica se o arquivo tem a extensão correta

if not csv\_file.name.endswith('.csv'):

form.add\_error('file', 'Este não é um arquivo CSV válido.')

else:

# Processa o arquivo CSV

file\_data = csv\_file.read().decode('ISO-8859-1').splitlines()

reader = csv.DictReader(file\_data, delimiter=',') # Altere para ',' se necessário

for row in reader:

try:

Sensor.objects.create(

tipo=row['tipo'],

unidade\_medida=row['unidade\_medida'] if row['unidade\_medida'] else None,

latitude=float(row['latitude'].replace(',', '.')),

longitude=float(row['longitude'].replace(',', '.')),

localizacao=row['localizacao'],

responsavel=row['responsavel'] if row['responsavel'] else '',

status\_operacional=True if row['status\_operacional'] == 'True' else False,

observacao=row['observacao'] if row['observacao'] else '',

mac\_address=row['mac\_address'] if row['mac\_address'] else None

)

except KeyError as e:

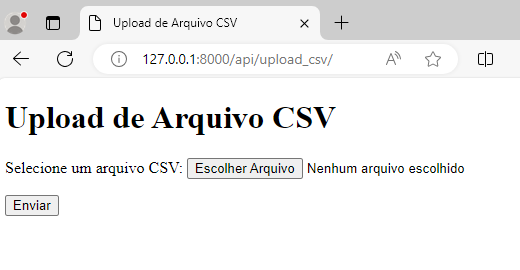
print(f"Chave não encontrada: {e} na linha: {row}") # Exibe o erro e a linha problemática

else:

form = CSVUploadForm()

return render(request, 'app\_smart/upload\_csv.html', {'form': form})

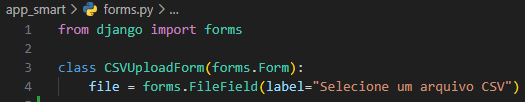
Crie um arquivo ***app\_smart/templates/upload\_csv.html*** para carregar o csv:







Crie um arquivo chamado ***app\_smart/forms.py*:**

****

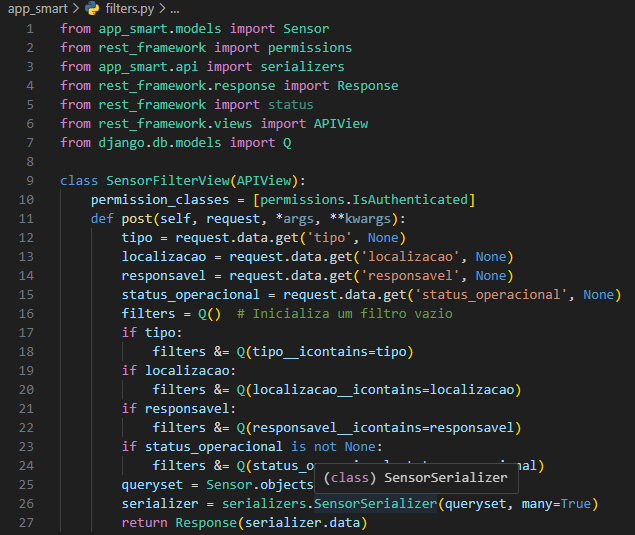
# 7 – Criando consulta pelo método POST

Para algumas situações pode ser mais adequado formular a consulta a uma API pelo método POST enviando um arquivo JSON no corpo da consulta.

Para isso será necessário criar uma rota específica para receber essa consulta. E é isso que faremos neste tópico.

Abra o arquivo **filters.py**

Acrescente as importações que ainda não estiverem:



Em **urls.py** do *app\_smart* acrescente a importação SensorFilterView que acabamos de criar: A linha completa fica assim:

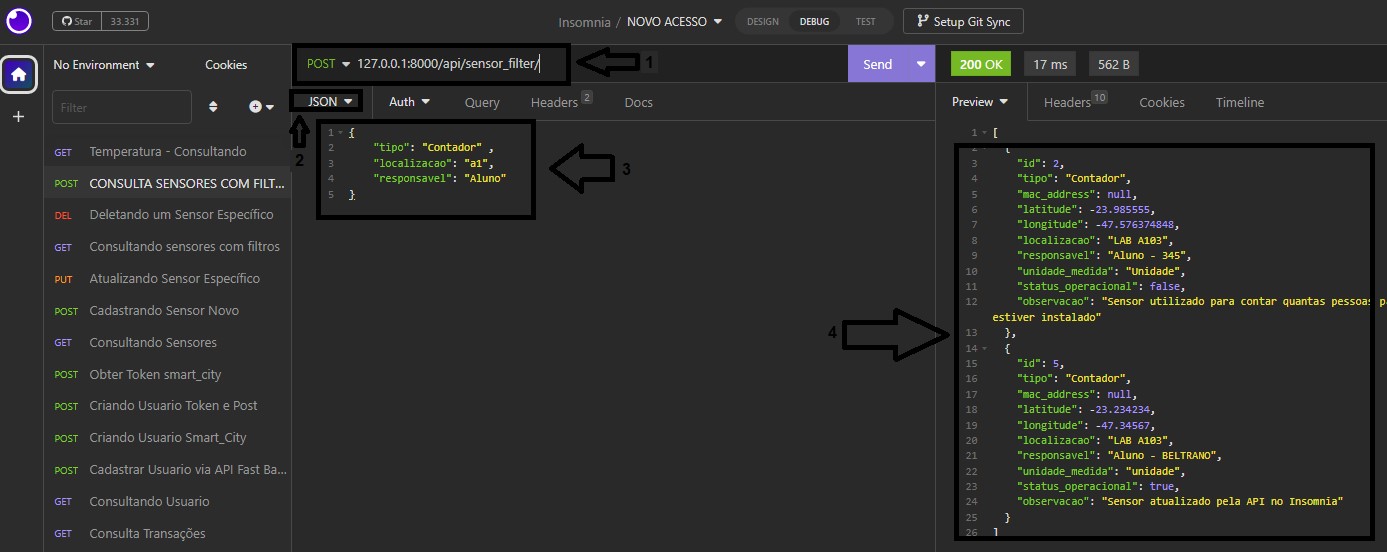


## 7.1 – Testando API sensores consultando pelo método POST via software Insomnia

Agora crie novas consultas no Insomnia para os filtros como parâmetros na URL como segue o exemplo:

Teste via Insomnia:

## 127.0.0.1:8000/api/sensor\_filter/



1. – Nova URL de consulta pelo método POST.
2. – Selecionar JSON como Body da mensagem.3 – Montar arquivo JSON com os dados da pesquisa

4 – Resultado da pesquisa.

# 8 – Criando *models* para armazenar dados dos sensores

Para armazenar os dados dos sensores e disponibilizá-los através de APIs, precisamos criar as tabelas correspondentes. Para cada tipo de sensor criaremos uma tabela fazendo com que a arquitetura do sistema fique mais adequada dado que um sensor pode produzir uma enorme quantidade de dados. Isso permitirá também consultas mais rápidas.

Assim abre o arquivo ***models.py***e acrescente o código que segue:

# Model para armazenar os dados de temperatura class TemperaturaData(models.Model): sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE) valor = models.FloatField() # Valor da temperatura em graus Celsius timestamp = models.DateTimeField(auto\_now\_add=True) # Momento da leitura def \_\_str\_\_(self): return f"Temperatura: {self.valor} °C - {self.timestamp}" # Model para armazenar os dados de umidade class UmidadeData(models.Model): sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE) valor = models.FloatField() # Valor da umidade relativa em % timestamp = models.DateTimeField(auto\_now\_add=True) # Momento da leitura def \_\_str\_\_(self): return f"Umidade: {self.valor}% - {self.timestamp}" # Model para armazenar os dados do contador class ContadorData(models.Model):

sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE) timestamp = models.DateTimeField(auto\_now\_add=True) # Momento da leitura def \_\_str\_\_(self): return f"Contagem - {self.timestamp}" # Model para armazenar os dados de luminosidade class LuminosidadeData(models.Model): sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE) valor = models.FloatField() # Valor da luminosidade em Lux

timestamp = models.DateTimeField(auto\_now\_add=True) # Momento da leitura def \_\_str\_\_(self): return f"Luminosidade: {self.valor} Lux - {self.timestamp}"

Depois de inserir o código no arquivo *models.py* execute o comando *makemigrations* e em seguida o comando *migrate*.

(amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> **py manage.py makemigrations**

Algumas linhas serão exibidas no terminal.

(amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> **py manage.py migrate.**

Observem que em todas as novas tabelas utilizamos um campo ***timestamp****.* Por default o Django assume como TIME\_ZONE=’UTC’ que significa Tempo Universal Coordenado.

Para que seja gravado no horário do Brasil (GMT-3) abra o arquivo *settings.py* do projeto e altera o opção TIME\_ZONE como segue e acrescente a linha USE\_TZ = True:

|  |  |
| --- | --- |
| **TIME\_ZONE = 'America/Sao\_Paulo'** | |
| USE\_TZ = True |  |

# Criando APIs para alimentar e consultar dados de sensores

Com o *model* criado para cada um dos tipos de sensores, vamos agora criar uma API para cada um desses models. Ou seja, teremos uma API para receber os dados dos sensores (ESP32 enviará os dados coletados através dessa API pelo método POST) e na mesma API poderemos fazer a consulta dos dados armazenados através do método GET.

# 9 - Criando API para os sensores de “*Temperatura*”

Assim como procedemos para a criação da API *Sensores*, depois de criado o model, faremos aqui com está APIpara o model *TemperaturaData.*

Ou seja, seguimos os passos:

*criar o arquivo serializers* > *criar o arquivo viewsets* > *criar as rotas de acesso.*

## 9.1 – Incluindo o model *TemperaturaData* no arquivo serializers.py

Abra o mesmo arquivo ***serializers.p*y** na pasta **api** em **smart\_city.**

Acrescente a importação do model:

from app\_smart.models import Sensor, TemperaturaData

Acrescente a classe *TemperaturaDataSerializer* como segue:

|  |  |  |  |
| --- | --- | --- | --- |
| class TemperaturaDataSerializer(serializers.ModelSerializer): | | | |
| class Meta: |  | | |
| model = TemperaturaData | | |  |
| fields = '\_\_all\_\_' | |  |

## 9.2– Incluindo a classe *TemperaturaDataViewSet* no arquivo viewsets.py

Abra o arquivo ***viewsets.py*** na pasta ***api*** em ***smart\_city****.*

Acrescente as importações:

from ..models import Sensor, TemperaturaData

E crie a classe *TemperaturaDataViewSet*como segue:

class TemperaturaDataViewSet(viewsets.ModelViewSet): queryset = TemperaturaData.objects.all()

serializer\_class = serializers.TemperaturaDataSerializer permission\_classes = [permissions.IsAuthenticated]

**9.3– Criando rotas de acesso para a API *Temperatura*** Abra o arquivo ***urls.py*** do ***app\_smart.***

Acrescente às importações *TemperaturaDataViewSet*:

from app\_smart.api.viewsets import (

CreateUserAPIViewSet,

SensorViewSet,

SensorFilterView,

TemperaturaDataViewSet

)

Acrescente a linha:

router.register(r'temperatura', TemperaturaDataViewSet)

Com isso a nossa rota para a API que acabamos de criar vai ser  [***http://127.0.0.1:8000/api/ temperatura***](http://127.0.0.1:8000/api/temperatura)

## 9.4 – Fazendo a carga de dados para o sensor de temperatura

No tópico 6, executamos o procedimento para fazer uma carga com os dados de Sensores, ou seja, onde os sensores serão instalados. Agora preparamos uma massa de dados expressiva para os sensores cadastrados simulando a geração de dados pelos sensores.

Agora vamos carregar essa massa em nossa BD, primeiramente paa os sensores do tipo ’Temperatura’ e criarmos uma consulta POST para esses dados.

Um arquivo chamado **temperatura\_data.csv** foi criado simulando os dados para os sensores de temperatura.

Crie um arquivo com no nome **load\_temperature.py** no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py)** e implemente o código que segue para importar os dados do arquivo CSV.

import csv

from datetime import datetime from dateutil import parser

import pytz import os import django

# Configuração do Django

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'smart\_city.settings') django.setup()

from app\_smart.models import TemperaturaData, Sensor def load\_temperature\_data(csv\_file\_path): print("Início da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S')) with open(csv\_file\_path, newline='', encoding='utf-8') as csvfile: reader = csv.DictReader(csvfile) line\_count = 0 for row in reader: sensor\_id = int(row['sensor\_id']) valor = float(row['valor'])

timestamp = parser.parse(row['timestamp']) # Usa dateutil para

analisar a data com fuso horário

sensor = Sensor.objects.get(id=sensor\_id)

TemperaturaData.objects.create(sensor=sensor, valor=valor, timestamp=timestamp) line\_count += 1 if line\_count % 10000 == 0: print(f"{line\_count} linhas processadas...")

print("Fim da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S'))

print(f"Dados carregados com sucesso de {csv\_file\_path}")

# Chame a função para carregar os dados do arquivo CSV load\_temperature\_data('dados/temperatura\_data.csv')

Na mesma pasta **dados** onde você colocou o arquivo *sensores.csv* no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py)** copie para esta pasta o arquivo **temperatura\_data.csv .**

Antes de fazer a carga desse arquivo no BD, verifique se há registros na tabela e se houver exclua os registros que inserimos como testes.

Abra o DB Browser e execute o seguinte comando:

### DELETE from app\_smart\_temperaturadata where id > 0;

|  |  |
| --- | --- |
| em seguida execute o comando | |
| **COMMIT** |  |

Verifique se todos os registros da tabela app\_smart\_sensor foram apagados.

Para que a carga possa começar novamente com ID=1, execute o comando que segue

#### DELETE FROM sqlite\_sequence WHERE name='app\_smart\_temperaturadata';

|  |  |
| --- | --- |
| em seguida execute o comando | |
| **COMMIT** |  |

Feche o BD Browser e retorne ao terminal.

**ATENÇÃO:**

**Esse arquivo de dados contém aproximadamente 600 mil linhas.**

**Dependendo da máquina pode demorar para executar. Portanto, antes de prosseguir confira se o seu “models” está igualmente criado como segue:**

**Model para armazenar os dados de temperatura** class TemperaturaData(models.Model): sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE) valor = models.FloatField() # Valor da temperatura em graus Celsius

#timestamp = models.DateTimeField(auto\_now\_add=True) # Momento da leitura timestamp = models.DateTimeField() def \_\_str\_\_(self): return f"Temperatura: {self.valor} °C - {self.timestamp}"

**Caso não esteja, não prossiga.**

**Faça as correções e execute os comandos makemigrations e migrate para que tenham efeito no BD. (ver tópico 8).**

**------------------------------------------------------------------------------------------------**

Estando tudo correto, vamos prosseguir:

|  |
| --- |
| (amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> |

Execute o comando como segue: **python load\_temperature.py**

Conforme pode ser observada no script, deve ser exibido a mensagem de *Início da Importação* com data e hora.

A cada 10.000 linhas será exibido a quantidade de linhas processadas.

Deve chegar até 590000 e depois exibir a mensagem *Fim da Importação* com data e hora e *‘Dados carregados com sucesso!!!’*



Confira no BD se os dados foram carregados.

**9.5 – Criando filtro para consulta pelo método GET para a API Temperatura.**

Já fizemos isso no tópico 5.3 para a API ‘*Sensores’.*

Agora faremos para nossa API de dados dos sensores de temperatura.

Abra o arquivo **filters.py.**

Acrescente a importação do model TemperaturaData.

from app\_smart.models import Sensor, TemperaturaData

Crie a classe *TemperaturaDataFilter* como segue:

class TemperaturaDataFilter(django\_filters.FilterSet): timestamp\_gte = django\_filters.DateTimeFilter(field\_name='timestamp', lookup\_expr='gte') timestamp\_lte = django\_filters.DateTimeFilter(field\_name='timestamp', lookup\_expr='lte') sensor = django\_filters.NumberFilter(field\_name='sensor')

valor\_gte = django\_filters.NumberFilter(field\_name='valor', lookup\_expr='gte') valor\_lte = django\_filters.NumberFilter(field\_name='valor', lookup\_expr='lte') class Meta: model = TemperaturaData fields = ['timestamp\_gte', 'timestamp\_lte', 'sensor', 'valor\_gte', 'valor\_lte']

Aqui temos uma nova expressão ***lookup\_exp=’gte’*** *e* ***lookup\_exp=’lte’ gte*** *“significa maior ou igual a” …* ***lte*** *“significa menor ou igual a” …*

*Observe que utilizamos esta expressão para os campos de* ***datas*** *e* ***valores****.*

Isso permitirá que nossas buscas se dêem por período de datas e por range de valores.

Por exemplo, poderemos especificar o período que desejamos que a busca seja feita.Como a massa de de dados fornecida foi para o período de 01 a 30-04-2024, podemos querer que a API retorne dados da temperatura de um certo sensor no período de 2024-04-10 a 202404-12, por exemplo.

A mesma coisa podemos fazer com o campo **valor** e especificar um range para o filtro. Por exemplo valor entre 22 e 24 graus. Agora abra o arquivo **viewsets.py** eacrescente a importação**: TemperaturaDataFilter**

**from app\_smart.api.filters import SensorFilter, TemperaturaDataFilter**

Na classe TemperaturaDataViewSet acrescente as seguintes linhas:

filter\_backends = [DjangoFilterBackend] filterset\_class = TemperaturaDataFilter

**9.6 – Criando consulta pelo método POST para a API Temperatura.**

Feito a carga dos dados de temperatura e o filtro pelo método GET, vamos criar a consulta a esses dados pelo método POST como segue:

Abra o arquivo **filters.py .** Acrescente a importação TemperaturaData se ainda não tiver feito:

from app\_smart.models import Sensor, TemperaturaData

Crie uma nova classe em **filters.py** como segue:

class TemperaturaFilterView(APIView): permission\_classes = [permissions.IsAuthenticated] def post(self, request, \*args, \*\*kwargs): sensor\_id = request.data.get('sensor\_id', None) valor\_gte = request.data.get('valor\_gte', None) valor\_lt = request.data.get('valor\_lt', None) timestamp\_gte = request.data.get('timestamp\_gte', None) timestamp\_lt = request.data.get('timestamp\_lt', None) filters = Q() # Inicializa um filtro vazio if sensor\_id:

filters &= Q(sensor\_id=sensor\_id) if valor\_gte: filters &= Q(valor\_\_gte=valor\_gte) if valor\_lt: filters &= Q(valor\_\_lt=valor\_lt) if timestamp\_gte:

filters &= Q(timestamp\_\_gte=timestamp\_gte) if timestamp\_lt: filters &= Q(timestamp\_\_lt=timestamp\_lt) queryset = TemperaturaData.objects.filter(filters)

serializer = serializers.TemperaturaDataSerializer(queryset, many=True) return Response(serializer.data)

Em **urls.py** do *app\_smart* acrescente a importação TemperaturaFilterView que acabamos de criar:

A linha completa fica assim:

Acrescente a seguinte rota na

sessão

urlpattern

s

path(

'api/temperatura\_filter/'

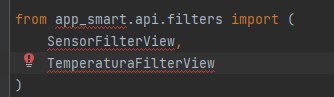
, TemperaturaFilterView.as\_view(),

name

=

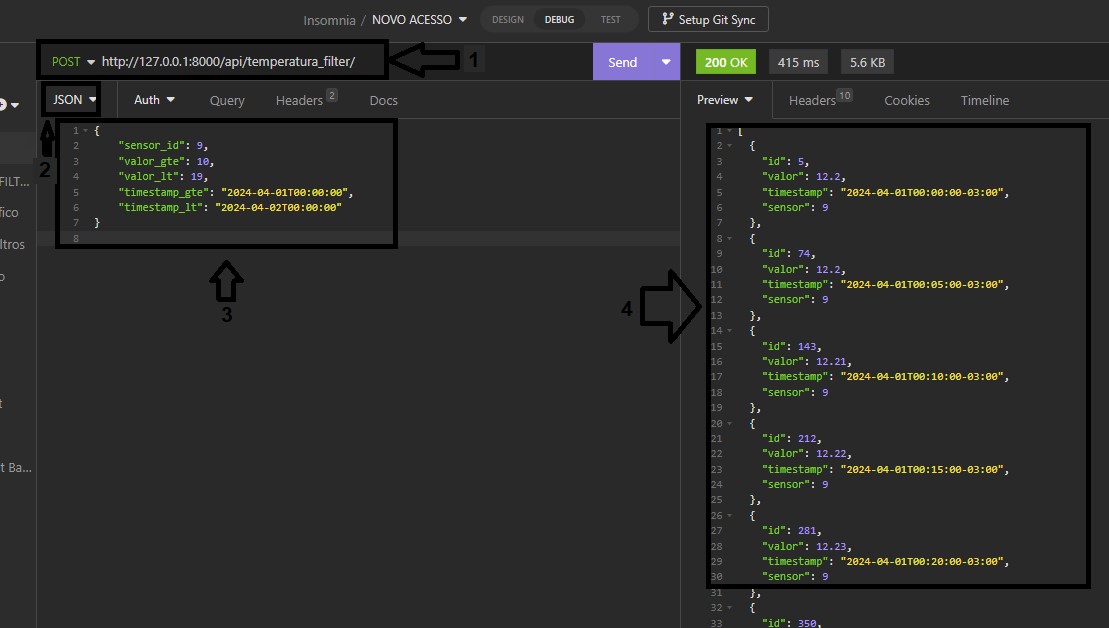
'temperatura\_filter'

)



**9.6.1 – Testando a consulta da API *temperatura\_filter*** **pelo Insomnia**

## 127.0.0.1:8000/api/temperatura\_filter/



1. – Nova URL de consulta pelo método POST.
2. – Selecionar JSON como Body da mensagem.3 – Montar arquivo JSON com os dados da pesquisa 4 – Resultado da pesquisa.

# 10 - Criando API para os dados do sensores de “*Umidade*”

Criaremos agora a API para os dados dos sensores de *Umidade* utilizando o model ***UmidadeData****.*

Assim, seguimos os passos:

*criar o arquivo serializers* > *criar o arquivo viewsets* > *criar as rotas de acesso.*

## 10.1 – Incluindo o model *UmidadeData* no arquivo serializers.py

Abra o mesmo arquivo ***serializers.p*y** na pasta **api** em **smart\_city.**

Acrescente a importação do model UmidadeData from app\_smart.models import Sensor, TemperaturaData, UmidadeData Acrescente a classe *UmidadeDataSerializer* como segue:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| class UmidadeDataSerializer(serializers.ModelSerializer): | | | | |
| class Meta: | |  | | |
| model = UmidadeData | | | |  |
|  | fields = '\_\_all\_\_' | |  |

## 10.2– Incluindo a classe *UmidadeDataViewSet* no arquivo viewsets.py

Abra o arquivo ***viewsets.py*** na pasta ***api*** em ***smart\_city****.*

Acrescente a importação do model UmidadeData:

from ..models import Sensor, TemperaturaData, UmidadeData

E crie a classe *UmidadeDataViewSet*como segue:

class UmidadeDataViewSet(viewsets.ModelViewSet): queryset = UmidadeData.objects.all()

serializer\_class = serializers.UmidadeDataSerializer permission\_classes = [permissions.IsAuthenticated]

## 10.3– Criando rotas de acesso para a API *Umidade*

Abra o arquivo ***urls.py*** do ***app\_smart.***

Acrescente às importações *UmidadeDataViewSet*:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| from app\_smart.api.viewsets import ( | | | | | |
| CreateUserAPIViewSet, | | | |  | |
| SensorViewSet, |  | | |
| SensorFilterView, | |  | |
| TemperaturaDataViewSet, | | | | |  |
| UmidadeDataViewSet, | | |  | |
|  | | |

)

Acrescente a linha:

router.register(r'umidade', UmidadeDataViewSet)

Com isso a nossa rota para a API que acabamos de criar vai ser  [***http://127.0.0.1:8000/api/ u***](http://127.0.0.1:8000/api/temperatura)***midade***

**10.4 – Fazendo a carga de dados para o sensor de umidade**

No tópico 9.4 fizemos a carga dos dados para os sensores de temperatura. Agora faremos o mesmo para os sensores de umidade.

Você tem disponibilizado o arquivo de dados chamado **umidade\_data.csv.** Copie-o para a pasta **dados** assim como foi feito com os arquivos de dados de sensores e temperatura.

Antes de realizar a carga. Assim como no tópico 9.4, verifique se a tabela está corretamente criada no BD e se não contem dados. Caso contenha dados execute os comandos para apagar os dados e depois o comando para apagar a “sequencia” de IDs na tabela.

### DELETE from app\_smart\_umidadedata where id > 0;

**COMMIT;**

#### DELETE FROM sqlite\_sequence WHERE name='app\_smart\_umidadedata';

**COMMIT**

Agora crie um arquivo com no nome **load\_umidade.py** no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py)** e implemente o código que segue para importar os dados do arquivo CSV.

import csv

from datetime import datetime from dateutil import parser

import pytz import os import django

# Configuração do Django

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'smart\_city.settings') django.setup()

from app\_smart.models import UmidadeData, Sensor def load\_umidade\_data(csv\_file\_path): print("Início da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S')) with open(csv\_file\_path, newline='', encoding='utf-8') as csvfile: reader = csv.DictReader(csvfile) line\_count = 0 for row in reader: sensor\_id = int(row['sensor\_id']) valor = float(row['valor']) timestamp =

parser.parse(row['timestamp']).astimezone(pytz.timezone('America/Sao\_Paulo')) sensor = Sensor.objects.get(id=sensor\_id)

UmidadeData.objects.create(sensor=sensor, valor=valor, timestamp=timestamp) line\_count += 1 if line\_count % 10000 == 0: print(f"{line\_count} linhas processadas...")

print("Fim da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S'))

print(f"Dados carregados com sucesso de {csv\_file\_path}")

# Chame a função para carregar os dados do arquivo CSV load\_umidade\_data('dados/umidade\_data.csv')

|  |
| --- |
| (amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> |

Execute o comando como segue: **python load\_umidade.py**

Exibirá no terminal a mensagem de *Início da Importação* com data e hora.

A cada 10.000 linhas será exibido a quantidade de linhas processadas.

Deve chegar a aproximadamente 160000 e depois exibir a mensagem *Fim da Importação* com data e hora e *‘Dados carregados com sucesso!!!’*



Confira no BD se os dados foram carregados.

**10.5 – Criando consulta pelo método POST para a API Umidade.**

Nesta API não faremos os filtros pelo método GET. Caso você deseje fazer também os filtros pelo método GET, siga os passos do tópico 9.5 alterando os nomes das classes e model.

Aqui, faremos diretamente os filtros pelo método POST como segue:

Abra o arquivo **filters.py .**

Acrescente a importação UmidadeData se ainda não tiver feito:

from app\_smart.models import Sensor, TemperaturaData, UmidadeData

Crie uma nova classe em **filters.py** como segue:

class UmidadeFilterView(APIView): permission\_classes = [permissions.IsAuthenticated] def post(self, request, \*args, \*\*kwargs): sensor\_id = request.data.get('sensor\_id', None) valor\_gte = request.data.get('valor\_gte', None) valor\_lt = request.data.get('valor\_lt', None) timestamp\_gte = request.data.get('timestamp\_gte', None) timestamp\_lt = request.data.get('timestamp\_lt', None)

filters = Q() # Inicializa um filtro vazio if sensor\_id:

filters &= Q(sensor\_id=sensor\_id) if valor\_gte:

filters &= Q(valor\_\_gte=valor\_gte) if valor\_lt:

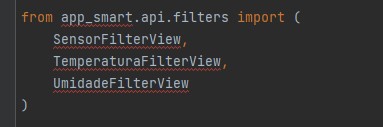
filters &= Q(valor\_\_lt=valor\_lt) if timestamp\_gte:

filters &= Q(timestamp\_\_gte=timestamp\_gte) if timestamp\_lt: filters &= Q(timestamp\_\_lt=timestamp\_lt) queryset = UmidadeData.objects.filter(filters)

serializer = serializers.UmidadeDataSerializer(queryset, many=True) return Response(serializer.data)

Em **urls.py** do *app\_smart* acrescente a importação UmidadeFilterView que acabamos de criar:

A linha completa fica assim:



Acrescente a seguinte rota na sessão urlpatterns

path('api/umidade\_filter/', UmidadeFilterView.as\_view(), name='umidade\_filter'),



**10.6 – Testando a consulta da API *umidade\_filter*** **pelo Insomnia.**

**127.0.0.1:8000/api/umidade\_filter/**

Utilize o seguinte JSON como exemplo para a busca:

{

"sensor\_id": 2,

"valor\_gte": 10,

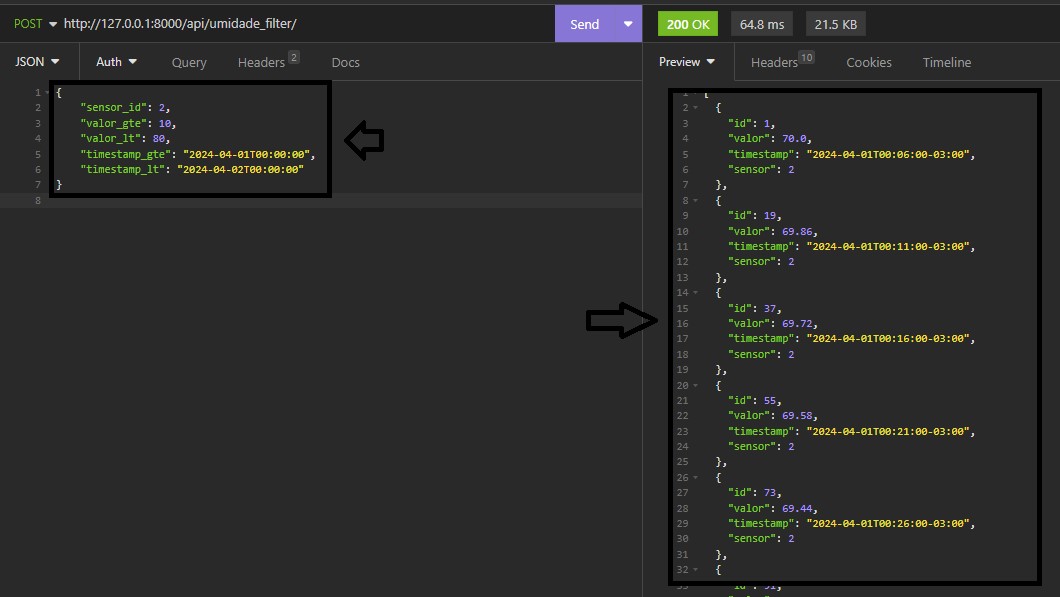
"valor\_lt": 80,

"timestamp\_gte": "2024-04-01T00:00:00",

"timestamp\_lt": "2024-04-02T00:00:00"

}

Resultado da consulta na imagem que segue:



# 11 - Criando API para os dados do sensores de “Luminosidade”

Criaremos agora a API para os dados dos sensores de Luminosidade utilizando o model ***LuminosidadeData****.*

Assim, seguimos os passos:

*criar o arquivo serializers* > *criar o arquivo viewsets* > *criar as rotas de acesso > criar filtro pelo método POST.*

## 11.1 – Incluindo o model *LuminosidadeData* no arquivo serializers.py

Abra o mesmo arquivo ***serializers.p*y** na pasta **api** em **smart\_city.**

Acrescente a importação do model LuminosidadeData

|  |  |
| --- | --- |
| from app\_smart.models import Sensor, TemperaturaData, UmidadeData, | |
| LuminosidadeData |  |
|  |

Acrescente a classe *LuminosidadeDataSerializer* como segue:

|  |  |  |  |
| --- | --- | --- | --- |
| class LuminosidadeDataSerializer(serializers.ModelSerializer): | | | |
| class Meta: |  | | |
| model = LuminosidadeData | | |  |
| fields = '\_\_all\_\_' | |  |

## 11.2– Incluindo a classe *UmidadeDataViewSet* no arquivo viewsets.py

Abra o arquivo ***viewsets.py*** na pasta ***api*** em ***smart\_city****.* Acrescente a importação do model LuminosidadeData: from ..models import Sensor, TemperaturaData, UmidadeData, LuminosidadeData

E crie a classe *LuminosidadeDataViewSet*como segue:

class LuminosidadeDataViewSet(viewsets.ModelViewSet): queryset = LuminosidadeData.objects.all()

serializer\_class = serializers.LuminosidadeDataSerializer permission\_classes = [permissions.IsAuthenticated]

**11.3– Criando rotas de acesso para a API *Umidade*** Abra o arquivo ***urls.py*** do ***app\_smart.***

Acrescente às importações *LuminosidadeDataViewSet*:

from app\_smart.api.viewsets import (

CreateUserAPIViewSet,

SensorViewSet,

SensorFilterView,

TemperaturaDataViewSet,

UmidadeDataViewSet,

LuminosidadeDataViewSet,

)

Acrescente a linha:

router.register(r'luminosidade', LuminosidadeDataViewSet)

Com isso a nossa rota para a API que acabamos de criar vai ser  [***http://127.0.0.1:8000/api/ l***](http://127.0.0.1:8000/api/temperatura)***uminosidade***

**11.4 – Fazendo a carga de dados para os sensores de Luminosidade**

Assim como foi feito nos tópicos 9.4 e 10.4, faremos o mesmo para os sensores de luminosidade.

Você tem disponibilizado o arquivo de dados chamado

**luminosidade\_data.csv.** Copie-o para a pasta **dados** assim como foi feito com os arquivos de dados de sensores e temperatura.

Antes de realizar a carga, verifique se a tabela está corretamente criada no BD e se não contem dados.

Caso contenha dados execute os comandos para apagar os dados e depois o comando para apagar a “sequencia” de IDs na tabela.

### DELETE from app\_smart\_luminosidadedata where id > 0;

**COMMIT;**

#### DELETE FROM sqlite\_sequence WHERE name='app\_smart\_luminosidadedata';

**COMMIT**

Agora crie um arquivo com no nome **load\_luminosidade.py** no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py)** e implemente o código que segue para importar os dados do arquivo CSV.

import csv

from datetime import datetime from dateutil import parser

import pytz import os import django

# Configuração do Django

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'smart\_city.settings') django.setup()

from app\_smart.models import LuminosidadeData, Sensor def load\_luminosidade\_data(csv\_file\_path): print("Início da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S')) with open(csv\_file\_path, newline='', encoding='utf-8') as csvfile: reader = csv.DictReader(csvfile) line\_count = 0 for row in reader: sensor\_id = int(row['sensor\_id']) valor = float(row['valor'])

timestamp = parser.parse(row['timestamp']) # Usa dateutil para

analisar a data com fuso horário

sensor = Sensor.objects.get(id=sensor\_id)

LuminosidadeData.objects.create(sensor=sensor, valor=valor, timestamp=timestamp) line\_count += 1 if line\_count % 10000 == 0: print(f"{line\_count} linhas processadas...")

print("Fim da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S'))

print(f"Dados carregados com sucesso de {csv\_file\_path}")

# Chame a função para carregar os dados do arquivo CSV load\_luminosidade\_data('dados/luminosidade\_data.csv')

|  |
| --- |
| (amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> |

Execute o comando como segue: **python load\_luminosidade.py**

Exibirá no terminal a mensagem de *Início da Importação* com data e hora.

A cada 10.000 linhas será exibido a quantidade de linhas processadas.

Deve chegar a aproximadamente 100.000 e depois exibir a mensagem *Fim da Importação* com data e hora e *‘Dados carregados com sucesso!!!’*



Confira no BD se os dados foram carregados.

**11.5 – Criando consulta pelo método POST para a API Luminosidade.**

Assim como fizemos na API de *Umidade,* criaremos nesta API apenas o método POST para filtro.

Abra o arquivo **filters.py .**

Acrescente a importação LuminosidadeData se ainda não tiver feito:

|  |  |
| --- | --- |
| from app\_smart.models import Sensor, TemperaturaData, UmidadeData, | |
| LuminosidadeData |  |

Crie uma nova classe chamada LuminosidadeFilterView em **filters.py** como segue:

class LuminosidadeFilterView(APIView): permission\_classes = [permissions.IsAuthenticated] def post(self, request, \*args, \*\*kwargs): sensor\_id = request.data.get('sensor\_id', None) valor\_gte = request.data.get('valor\_gte', None) valor\_lt = request.data.get('valor\_lt', None) timestamp\_gte = request.data.get('timestamp\_gte', None) timestamp\_lt = request.data.get('timestamp\_lt', None) filters = Q() # Inicializa um filtro vazio if sensor\_id:

filters &= Q(sensor\_id=sensor\_id) if valor\_gte:

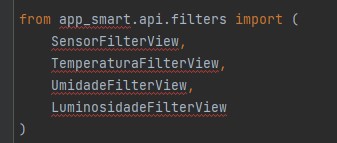
filters &= Q(valor\_\_gte=valor\_gte) if valor\_lt: filters &= Q(valor\_\_lt=valor\_lt) if timestamp\_gte:

filters &= Q(timestamp\_\_gte=timestamp\_gte) if timestamp\_lt: filters &= Q(timestamp\_\_lt=timestamp\_lt) queryset = LuminosidadeData.objects.filter(filters)

serializer = serializers.LuminosidadeDataSerializer(queryset, many=True) return Response(serializer.data)

Em **urls.py** do *app\_smart* acrescente a importação LuminosidadeFilterView que acabamos de criar:

A linha completa fica assim agora :



Acrescente a seguinte rota na sessão urlpatterns

path('api/luminosidade\_filter/', LuminosidadeFilterView.as\_view(),

name='luminosidade\_filter'),

A sessão ***urlpattern*** deve ficar como segue:



**11.6 – Testando a consulta da API *luminosidade\_filter*** **pelo Insomnia.**

**127.0.0.1:8000/api/luminosidade\_filter/**

Utilize o seguinte JSON como exemplo para a busca:

{

"sensor\_id": 19,

"valor\_gte": 10,

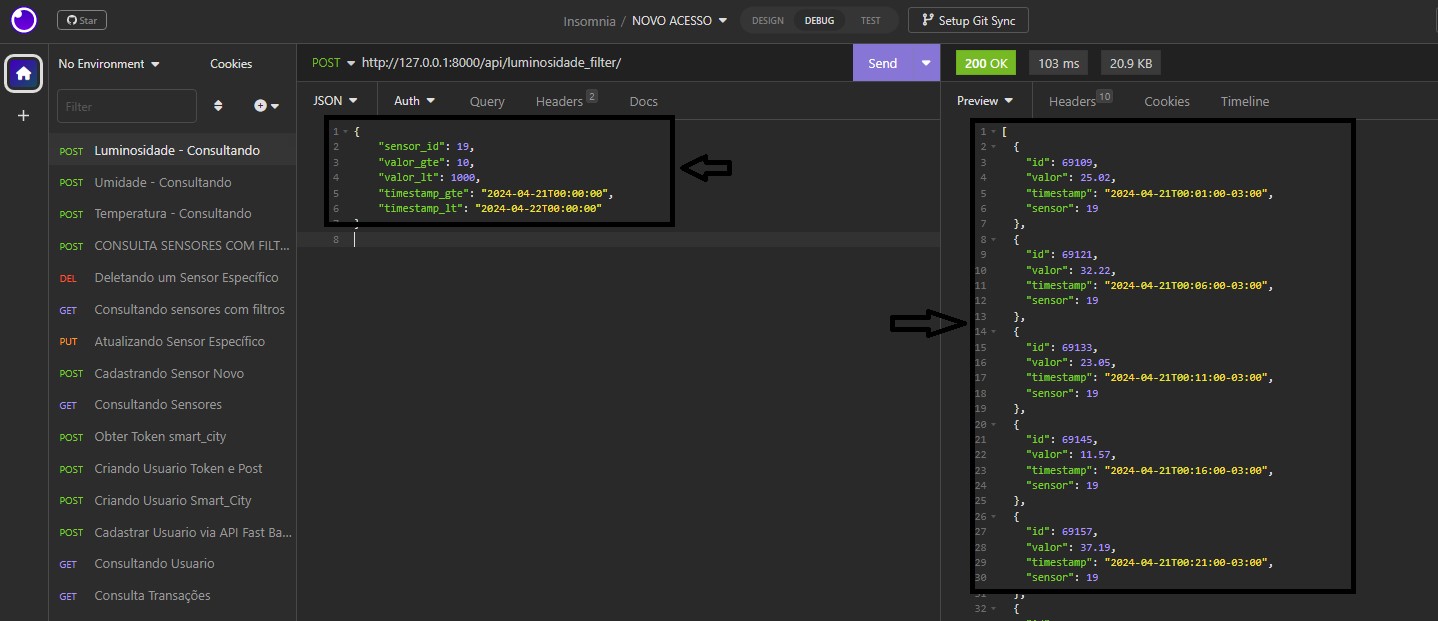
"valor\_lt": 1000,

"timestamp\_gte": "2024-04-21T00:00:00",

"timestamp\_lt": "2024-04-22T00:00:00"

}

Resultado da consulta na imagem que segue:



# 12 - Criando API para os dados do sensores do tipo “Contador”

O sensor do tipo ”Contador” registra apenas a data e horário de seu acionamento. A consulta portanto é relevante retornar, além dos registros específicos, uma contagem de todos os registros.

Assim, seguimos os passos:

*criar o arquivo serializers* > *criar o arquivo viewsets* > *criar as rotas de acesso > criar filtro pelo método POST.*

## 12.1 – Incluindo o model ContadorD*ata* no arquivo serializers.py

Abra o mesmo arquivo ***serializers.p*y** na pasta **api** em **smart\_city.**

Acrescente a importação do model ContadorData

from app\_smart.models import Sensor, TemperaturaData, UmidadeData,

LuminosidadeData, ContadorData

Acrescente a classe *ContadorDataSerializer* como segue:

class LuminosidadeDataSerializer(serializers.ModelSerializer): class Meta: model = ContadorData fields = '\_\_all\_\_'

## 12.2– Incluindo a classe *ContadorDataViewSet* no arquivo viewsets.py

Abra o arquivo ***viewsets.py*** na pasta ***api*** em ***smart\_city****.*

Acrescente a importação do model ContadorData:

from ..models import Sensor, TemperaturaData, UmidadeData, LuminosidadeData, ContadorData

E crie a classe *ContadorDataViewSet*como segue:

class ContadorDataViewSet(viewsets.ModelViewSet): queryset = ContadorData.objects.all()

serializer\_class = serializers.ContadorDataSerializer permission\_classes = [permissions.IsAuthenticated]

**12.3– Criando rotas de acesso para a API Contador** Abra o arquivo ***urls.py*** do ***app\_smart.***

Acrescente às importações *ContadorDataViewSet*:

from app\_smart.api.viewsets import (

CreateUserAPIViewSet,

SensorViewSet,

SensorFilterView,

TemperaturaDataViewSet,

UmidadeDataViewSet,

LuminosidadeDataViewSet,

ContadorDataViewSet,

)

Acrescente a linha:

router.register(r'contador', ContadorDataViewSet)

Com isso a nossa rota para a API que acabamos de criar vai ser  [***http://127.0.0.1:8000/api/ contador***](http://127.0.0.1:8000/api/contador)

**12.4 – Fazendo a carga de dados para os sensores tipo ‘Contador’**

Assim como foi feito nos tópicos 9.4 e 10.4, e 11.4 faremos o mesmo para os sensores do tipo ‘Contador’.

Você tem disponibilizado o arquivo de dados chamado

**contador\_data.csv.** Copie-o para a pasta **dados** assim como foi feito com os arquivos de dados de sensores e temperatura.

Antes de realizar a carga, verifique se a tabela está corretamente criada no BD e se não contem dados.

Caso contenha dados execute os comandos para apagar os dados e depois o comando para apagar a “sequencia” de IDs na tabela.

### DELETE from app\_smart\_contadordata where id > 0;

**COMMIT;**

#### DELETE FROM sqlite\_sequence WHERE name='app\_smart\_contadordata';

**COMMIT**

Agora crie um arquivo com no nome **load\_contador.py** no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py)** e implemente o código que segue para importar os dados do arquivo CSV.

import csv

from datetime import datetime from dateutil import parser

import pytz import os import django

# Configuração do Django

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'smart\_city.settings') django.setup()

from app\_smart.models import ContadorData, Sensor def load\_contador\_data(csv\_file\_path): print("Início da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S')) with open(csv\_file\_path, newline='', encoding='utf-8') as csvfile: reader = csv.DictReader(csvfile) line\_count = 0 for row in reader: sensor\_id = int(row['sensor\_id'])

timestamp = parser.parse(row['timestamp']) # Usa dateutil para analisar a data com fuso horário

sensor = Sensor.objects.get(id=sensor\_id)

ContadorData.objects.create(sensor=sensor, timestamp=timestamp)

line\_count += 1 if line\_count % 10000 == 0: print(f"{line\_count} linhas processadas...")

print("Fim da importação:", datetime.now().strftime('%Y-%m-%d %H:%M:%S'))

print(f"Dados carregados com sucesso de {csv\_file\_path}")

# Chame a função para carregar os dados do arquivo CSV load\_contador\_data('dados/contador\_data.csv')

|  |
| --- |
| (amb\_virtual) PS C:\PWBE\Sprint\_3\smart\_city> |

Execute o comando como segue: **python load\_contador.py**

Exibirá no terminal a mensagem de *Início da Importação* com data e hora.

A cada 10.000 linhas será exibido a quantidade de linhas processadas.

Deve chegar a aproximadamente 18.000 e depois exibir a mensagem *Fim da Importação* com data e hora e *‘Dados carregados com sucesso!!!’*



Confira no BD se os dados foram carregados.

**12.5 – Criando consulta pelo método POST para a API Contador.**

Assim como fizemos na API de *Umidade,* criaremos nesta API apenas o método POST para filtro. Porém, neste caso além de retornar os registros selecionados com o filtro, trará também uma contagem da quantidade de registros retornados.

Abra o arquivo **filters.py .**

Acrescente a importação ContadorData se ainda não tiver feito:

|  |  |
| --- | --- |
| from app\_smart.models import Sensor | |
| LuminosidadeData, ContadorData |  |

|  |  |  |
| --- | --- | --- |
| TemperaturaData | , | UmidadeData, |

,

Crie uma nova classe chamada ContadorFilterView em **filters.py** como segue:

class ContadorFilterView(APIView): permission\_classes = [permissions.IsAuthenticated] def post(self, request, \*args, \*\*kwargs): sensor\_id = request.data.get('sensor\_id', None) timestamp\_gte = request.data.get('timestamp\_gte', None) timestamp\_lt = request.data.get('timestamp\_lt', None)

filters = Q() # Inicializa um filtro vazio if sensor\_id: filters &= Q(sensor\_id=sensor\_id) if timestamp\_gte:

filters &= Q(timestamp\_\_gte=timestamp\_gte) if timestamp\_lt: filters &= Q(timestamp\_\_lt=timestamp\_lt) queryset = ContadorData.objects.filter(filters)

count = queryset.count()

serializer = serializers.ContadorDataSerializer(queryset, many=True)

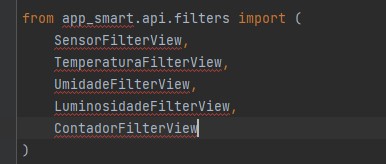
response\_data = { 'count': count,

'results': serializer.data

} return Response(response\_data)

Em **urls.py** do *app\_smart* acrescente a importação ContadorFilterView que acabamos de criar:

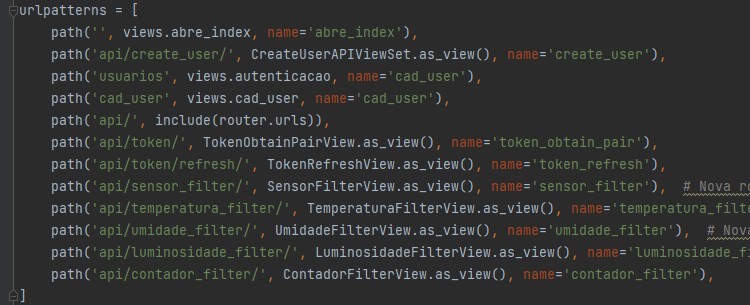
A linha completa fica assim agora :



Acrescente a seguinte rota na sessão urlpatterns

path('api/contador\_filter/', ContadorFilterView.as\_view(), name='contador\_filter'),

A sessão ***urlpattern*** deve ficar como segue:



**12.6 – Testando a consulta da API *contador\_filter*** **pelo Insomnia.**

**127.0.0.1:8000/api/contador\_filter/**

Utilize o seguinte JSON como exemplo para a busca:

**{**

**"sensor\_id": 70,**

**"timestamp\_gte": "2024-04-01T00:00:00",**

**"timestamp\_lt": "2024-04-30T00:00:00" }**

O resultado dessa consulta é o que segue:

