

A partir de agora criaremos um novo 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 juntamente com o Banco de Dados da aplicação.

## 1 – Criando o projeto “**smart\_city**”

Utilizando o PyCharm criaremos o novo projeto clicando “**File > New Project**”.

Agora você pode criar um novo ambiente virtual ou escolher um ambiente virtual já existente.

Verifique se neste ambiente virtual você já tem o Django instalado. Caso não tenha execute o comando

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> > pip install django
```

Agora nesse ambiente virtual onde você irá desenvolver este projeto. Faça a instalação do Django Rest Framework executando o seguinte comando no terminal:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> > pip install djangorestframework
```

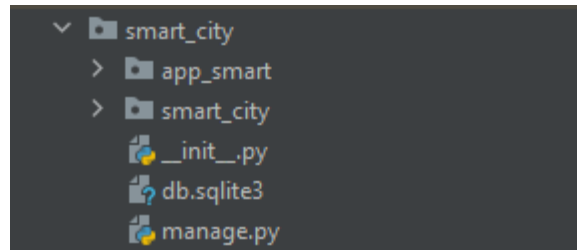
Feitos os passos acima, vamos criar o projeto **smart\_city** no Django com o comando:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> > django-admin startproject smart_city
```

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

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> py manage.py startapp app_smart
```

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



Agora acesse o arquivo **settings.py** que está dentro da pasta `smart_city` do projeto e na sessão `INSTALLED_APPS` acrescente as seguintes linhas:

```
'rest_framework',
'app_smart'
```

A sessão `INSTALLED_APPS` deve ficar assim:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'app_smart'
]
```

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

```
(amb virtual) PS C:\PWBE\Sprint 3\smart city> py manage.py migrate
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

Confira com o DB Browser se as tabelas do Django foram criadas.

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

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> py manage.py createsuperuser
```

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

```
username = 'smart_user'  
password = '123456'
```

Criamos agora a rota de inicialização da aplicação editando o arquivo **urls.py** do projeto. O arquivo deve ficar assim:

```
from django.contrib import admin  
from django.urls import path, include  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('', include('app_smart.urls'))  
]
```

Agora vamos criar o arquivo **urls.py** do **app\_smart** como segue:

```
from django.urls import path, include  
from . import views  
urlpatterns = [  
    path('', views.abre_index, name='abre_index'),  
]
```

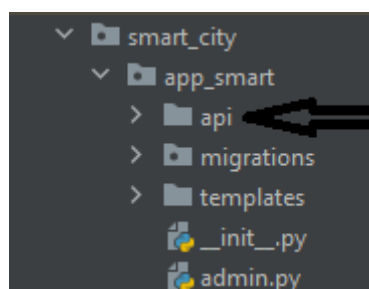
Como criamos no arquivo acima a chamada da função ‘*abre\_index*’ vamos criar essa função em **views.py**:

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def abre_index(request):
    mensagem = "OLÁ TURMA, SEJAM FELIZES SEMPRE!"
    return HttpResponse(mensagem)
```

## 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 novo arquivo chamado **serializers.py** e implemente o seguinte código:

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

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'password'] # Adicione outros campos se necessário
        extra_kwargs = {'password': {'write only': True}}
```

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:

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 novo arquivo chamado **viewsets.py** e implemente o seguinte código:

```
from django.contrib.auth.models import User
from rest_framework import generics, permissions
from app_smart.api import serializers # app_smart é o app criado.
from rest_framework.response import Response
from rest_framework import status

class CreateUserAPIView(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = serializers.UserSerializer
    permission_classes = [permissions.IsAdminUser]

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

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 seguida cria 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:

```
from app_smart.api.viewsets import CreateUserAPIViewSet
```

```
path('api/create_user/', CreateUserAPIViewSet.as_view(), name='create_user'),
```

Isso fará com que nossa API já esteja disponível.

## 2.4 – Testando a API

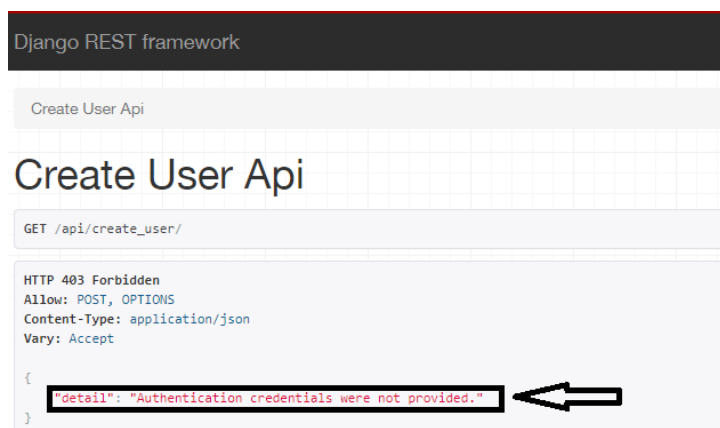
### 2.4.1 – Testando a API via admin do Django

Execute o servidor com o comando:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> py manage.py runserver
```

No Browser acesso o endereço da API

**http://127.0.0.1:8000/api/create\_user**



Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

Se a página anterior foi exibida, sinal de que sua API já está funcionando. Porém ela apresenta uma mensagem de que as credencias não foram fornecidas. Isto acontece porque em **viewsets.py** foi criado a seguinte linha `permission_classes = [permissions.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.

## Create User Api

OPTIONS

GET /api/create\_user/

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "detail": "Method \"GET\" not allowed."
}
```

Raw data

HTML form

Username

Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.

Email address

Password

POST

Retire o comentário da linha `permission_classes = [permissions.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 parâmetros da credencial de um super usuário, você pode utilizar os comandos “*Fetch*” e “*Axios*”. A seguir temos um exemplo de cada enviando como credenciais o *superusuário* criado quando da criação do projeto que foi **username=smart\_user** e **password=123456**

### Exemplo Fetch:

```
<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);
    fetch('/api/create_user/', {
        method: 'POST',
        headers: {
            'Authorization': 'Basic ' + btoa('smart_user:123456')
        },
        body: formData
    })
    .then(response => {
        if (!response.ok) {
            throw new Error('Erro ao cadastrar usuário');
        }
        alert('Usuário cadastrado com sucesso!');
    })
    .catch(error => {
        console.error('Erro:', error);
        alert('Erro ao cadastrar usuário');
    });
});
</script>
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

## Exemplo Axios:

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<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('/api/create_user/', formData, {
        headers: {
            'Authorization': 'Basic ' + btoa('smart_user:123456')
        }
    })
    .then(response => {
        alert('Usuário cadastrado com sucesso!');
    })
    .catch(error => {
        console.error('Erro:', error);
        alert('Erro ao cadastrar usuário');
    });
});
</script>
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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

Após ter criado um novo usuário, abra o DB Browser e verifique os dados do usuário que você cadastrou. No campo ‘password’ observe que foi armazenado sem criptografia.

DB Browser for SQLite - C:\PWBE\Sprint\_3\smart\_city\db.sqlite3

Arquivo Editar Exibir Ferramentas Ajuda

Novo banco de dados Abrir banco de dados Escrever modificações Reverter modificações Abrir projeto Salvar projeto Anexar banco de dados

Estrutura do banco de dados Navegar dados Editar pragmas Executar SQL

Tabela: auth\_user

	id	password	last_login	is_superuser	username	last_name	email	is_staff	is_active	
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	pbkdf2_sha256\$720000\$hNwWdfnKaH24R5AGcR...	NULL	1	smart_user		israel@gmail.com	1	1	2024
2	2	123456	NULL	0	israel			0	1	2024

Vamos corrigir isso implementando a criptografia para este campo.

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

```
from django.contrib.auth.hashers import make_password

password = serializers.CharField(write_only=True)

def create(self, validated_data):
    # Criptografar a senha antes de salvar o usuário
    validated_data['password'] = make_password(validated_data['password'])
    return super().create(validated_data)
```

O arquivo ficará como segue:

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

class UserSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True) # Campo de senha para escrita apenas

    def create(self, validated_data):
        # Criptografar a senha antes de salvar o usuário
        validated_data['password'] = make_password(validated_data['password'])
        return super().create(validated_data)

    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'password'] # Adicione outros campos se necessário
        extra_kwargs = {'password': {'write_only': True}}
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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

DB Browser for SQLite - C:\PWBE\Sprint\_3\smart\_city\db.sqlite3

Arquivo Editar Exibir Ferramentas Ajuda

Novo banco de dados Abrir banco de dados Escrever modificações Reverter modificações Abrir projeto Salvar projeto Anexar banco de dados Fechar banco

Estrutura do banco de dados Navegar dados Editar pragmas Executar SQL

Tabela: auth\_user

	id	password	last_login	is_superuser	username	last_name	email	is_staff	is_active	date_joined
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	pbkdf2_sha256\$720000\$hNwWDFnKaH24R5AGcR...	NULL	1	smart_user		israel@gmail.com	1	1	2024-05-09 20:51:59
2	2	123456	NULL	0	israel			0	1	2024-05-09 21:02:22
3	3	123456	NULL	0	novo_user			0	1	2024-05-09 21:34:37
4	4	123456	NULL	0	user_rafael		israel@gmail.com	0	1	2024-05-14 18:49:36
5	5	123456	NULL	0	mateus_user		mateus.igs@gmail.com	0	1	2024-05-14 18:57:57
6	6	pbkdf2_sha256\$720000\$RSvUiloFuyhlaMQaHqTBl...			raf_mat		gomes_silva@gmail.com	0	1	2024-05-14 19:20:10

### 3 – Instalando a autenticação JWT na API

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

Acesse o terminal e execute o comando como segue:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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:

```
'rest_framework_simplejwt.token_blacklist',
```

A sessão `INSTALLED_APPS` deve ficar assim:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'app_smart',  
    'rest_framework_simplejwt.token_blacklist',  
]
```

A próxima sessão no mesmo arquivo *settings.py* não existe até então. Crie conforme segue:

```
REST_FRAMEWORK = {  
    'DEFAULT_AUTHENTICATION_CLASSES': (  
        'rest_framework_simplejwt.authentication.JWTAuthentication',  
        'rest_framework.authentication.SessionAuthentication',  
    ),  
}
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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:

```
SIMPLE_JWT = {  
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),  
    # Define o tempo de expiração do token JWT PAdrão é 5 minutos  
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),  
    # Define o tempo de expiração do refresh token Padrao é 30 dias  
}
```

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

```
from datetime import timedelta
```

Agora precisamos criar as URLs para obtenção dos Tokens de acesso. Abra o arquivo *urls.py* do seu *app\_smart* e acrescente as seguintes linhas:

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView
```

```
urlpatterns = [  
    ...  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),  
]
```

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

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python manage.py migrate
```

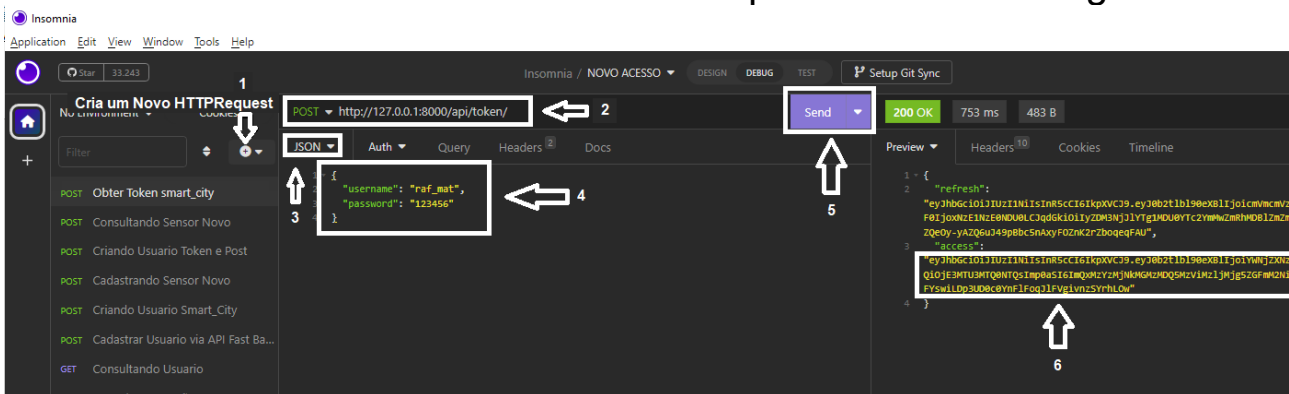
## 3.2 – Testando o acesso à API com Token

### 3.2.1 – Testando API utilizando o software Insomnia

Execute o servidor da API:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> py manage.py runserver
```

Abre o Insomnia e crie um novo 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/> e selecione a opção ‘POST’

3 – Selecione a opção ‘JSON’

4 – Monte um arquivo JSON com username e password conforme imagem. Utilize o *username=smart\_user* e o *password='123456'*

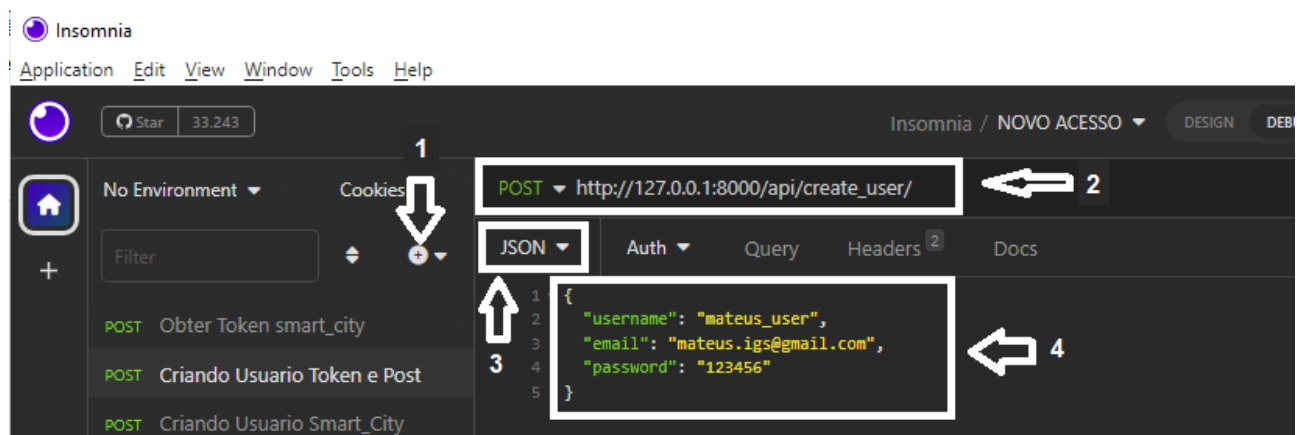
```
{  
    "username": "smart_user",  
    "password": "123456"  
}
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

5 – Clique em ‘Send’.

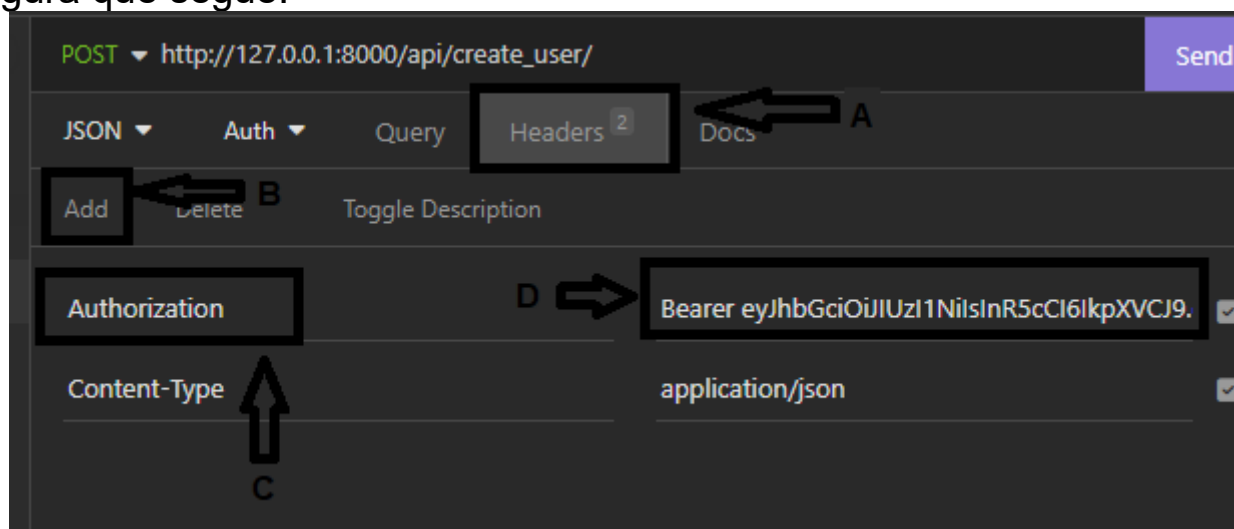
6 – Copie o ‘Token’ gerado.

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



Os passos 1 ao 4 se repetem como na obtenção do token.

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





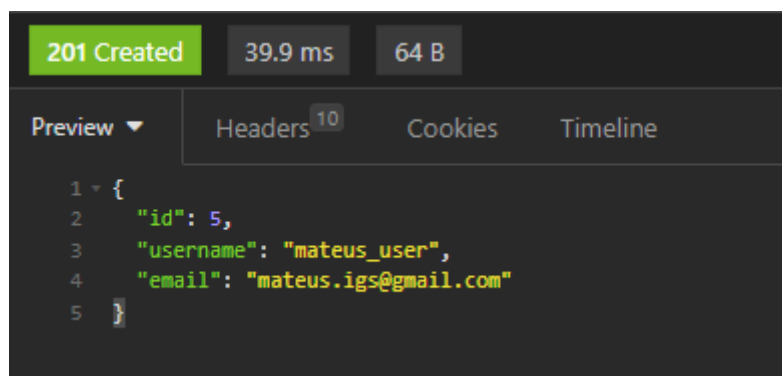
A – Selecione a aba “**Headers**”

B – Clique em ‘**Add**’. Abrirá a linha para criação de um novo cabeçalho

C – Na linha que abre, digite ‘*Authorization*’

D – 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:



## 4 – Criando uma API para cadastro de sensores

Para a criação de uma API devemos seguir os passos:

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

Para esta API ainda não criamos o Model que armazena os dados dos sensores. Então vamos fazer isso agora.

## 4.1 – Criando o model Sensor

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

```
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)
    #blank = True indica que o campo pode ser deixado em branco,
    #null = True indica que o campo pode ser gravado como null no banco de dados
    #as propriedades dizem que podem não ser preenchido tanto no formulario como no banco de dados
    unidade_medida = models.CharField(max_length=20, blank=True, null=True)
    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*.

```
(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.
```

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

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

Acrescente a importação do model

```
from app_smart.models import Sensor
```

E crie a classe **SensorSerializer** como segue:

```
class SensorSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Sensor  
        fields = '__all__' # Isso serializa todos os campos do modelo Sensor
```

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

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

Acrescente as importações

```
from ..models import Sensor #.. é para voltar dois níveis para encontrar o  
                             # arquivo models.py  
from rest_framework import viewsets
```

E crie a classe **SensorViewSet** como segue:

```
class SensorViewSet(viewsets.ModelViewSet):  
    queryset = Sensor.objects.all()  
    serializer_class = serializers.SensorSerializer  
    permission_classes = [permissions.IsAuthenticated]
```

## 4.4 – Criando rotas de acesso para a API

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

Acrescente as importações

```
from app_smart.api.viewsets import CreateUserAPIViewSet, SensorViewSet  
  
from rest_framework.routers import DefaultRouter
```

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 para os diferentes métodos HTTP com base nas ações definidas em ViewSets, simplificando a configuração das URLs da API.

Depois das importações acrescente as seguinte linhas:

```
router = DefaultRouter()  
router.register(r'sensores', SensorViewSet)
```

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 considerado como caractere 'escape' do Python, como por exemplo '\n' que criaria uma nova linha.

E na sessão `urlpatterns` acrescente a seguinte linha:

```
path('api/', include(router.urls)),
```

Com isso a nossa rota para a API que acabamos de criar vai ser <http://127.0.0.1:8000/api/sensores>

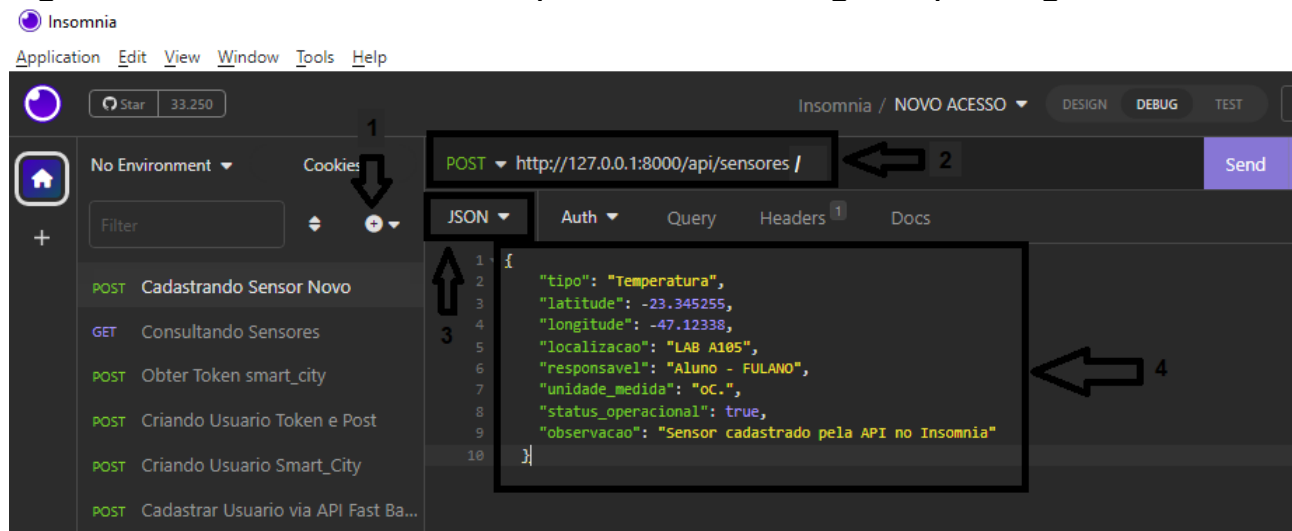
## 4.5 – Testando API utilizando o software Insomnia

Execute o servidor da API:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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 um 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/>’ e selecione a opção ‘POST’

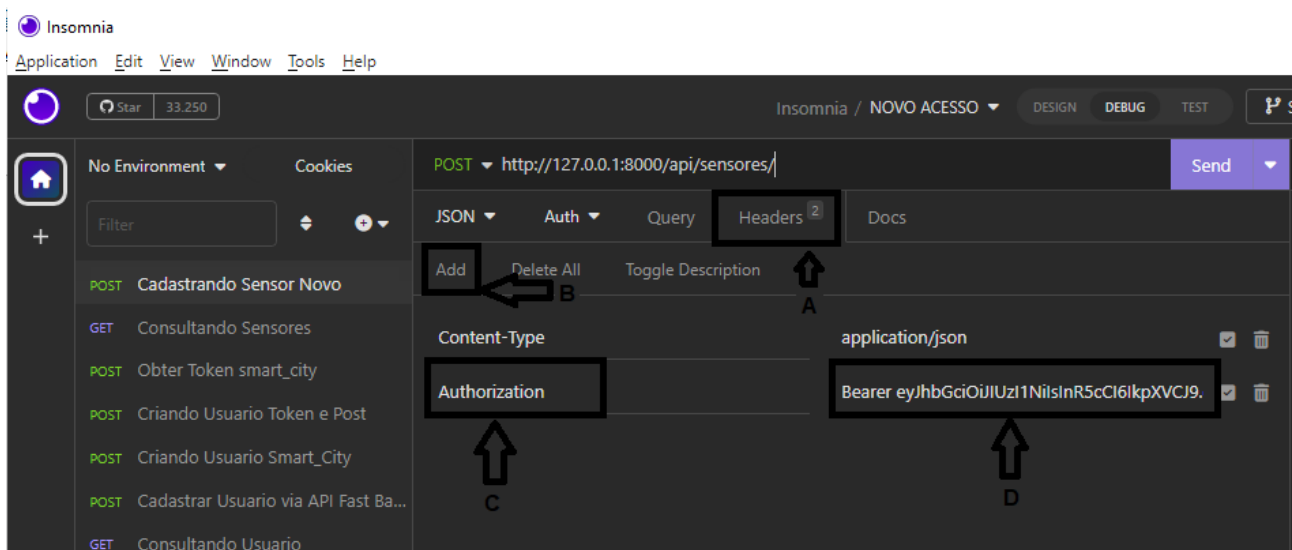
3 – Selecione a opção ‘JSON’

4 – Monte um arquivo JSON com os dados do sensor a ser cadastrado conforme segue:

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

```
{  
    "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"  
}
```

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



A – Selecione a aba “**Headers**”

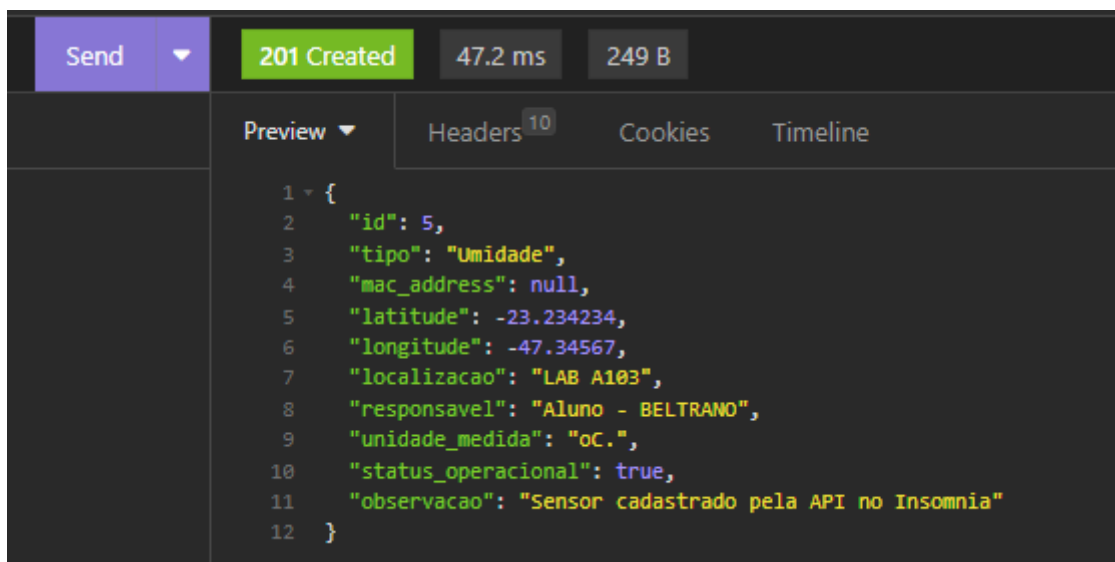
B – Clique em ‘**Add**’. Abrirá a linha para criação de um novo cabeçalho

C – Na linha que abre, digite ‘**Authorization**’

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

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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



## 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:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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:

```
'django_filters',
```



## 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 que segue:

```
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 o nomes: *'João da Silva'* e outro como o nome *'Rafael e Silva'* cadastrados no campo *'responsavel'*.

Se na busca eu informar apenas *'silva'* os dois registro 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:

```
from app_smart.api.filters import SensorFilter
from django_filters.rest_framework import DjangoFilterBackend
```

E na classe `SensorViewSet` acrescente as seguintes linhas:

```
filter_backends = [DjangoFilterBackend]
filterset_class = SensorFilter
```

A classe `SensorViewSet` no arquivo `viewsets.py` fica assim:

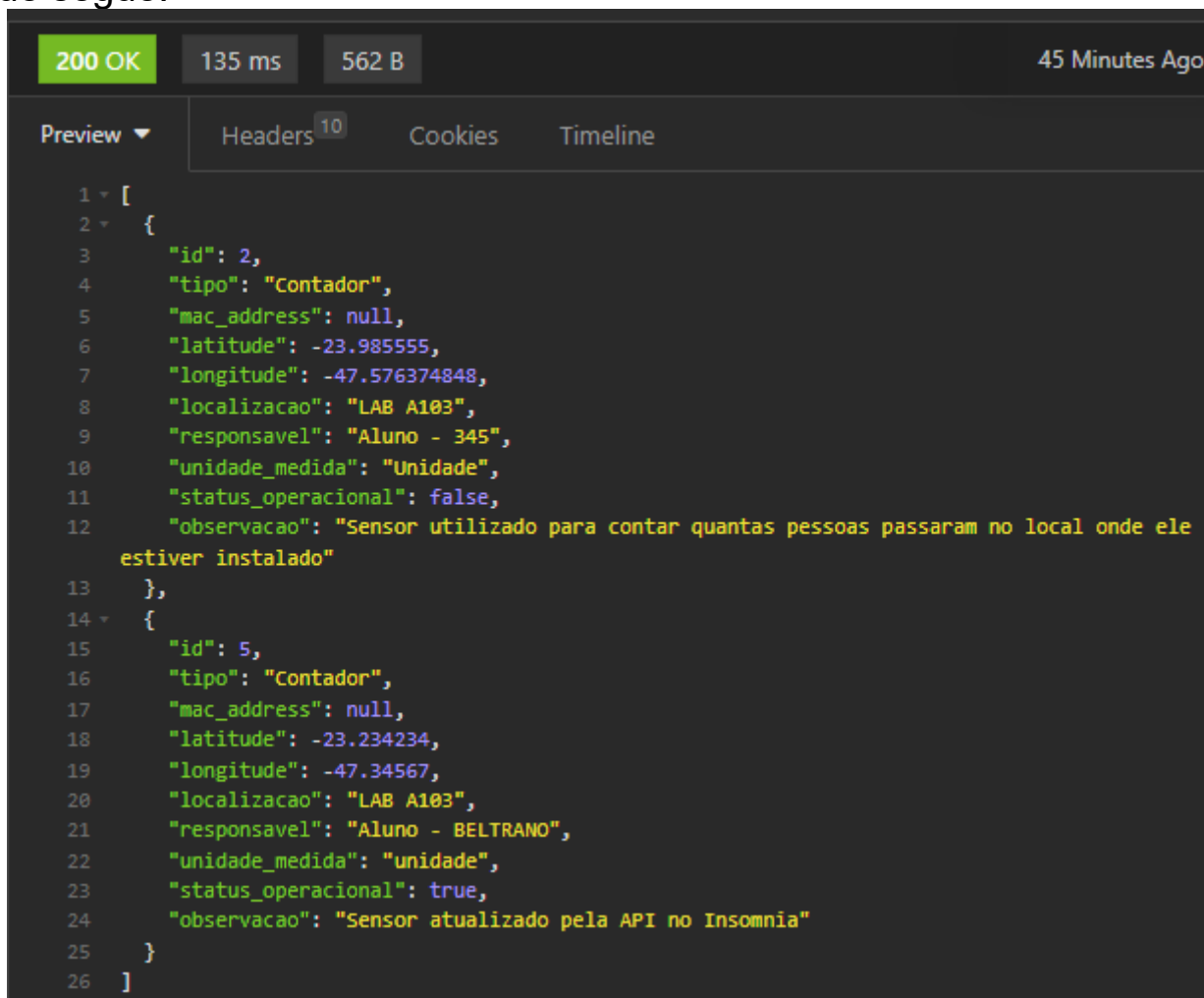
```
class SensorViewSet(viewsets.ModelViewSet):
    queryset = Sensor.objects.all()
    serializer_class = serializers.SensorSerializer
    permission_classes = [permissions.IsAuthenticated]
    filter_backends = [DjangoFilterBackend]
    filterset_class = SensorFilter
```

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

Agora crie novas 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:



```
200 OK 135 ms 562 B 45 Minutes Ago
Preview Headers 10 Cookies Timeline
1 [
2   {
3     "id": 2,
4     "tipo": "Contador",
5     "mac_address": null,
6     "latitude": -23.985555,
7     "longitude": -47.576374848,
8     "localizacao": "LAB A103",
9     "responsavel": "Aluno - 345",
10    "unidade_medida": "Unidade",
11    "status_operacional": false,
12    "observacao": "Sensor utilizado para contar quantas pessoas passaram no local onde ele
    estiver instalado"
13  },
14  {
15    "id": 5,
16    "tipo": "Contador",
17    "mac_address": null,
18    "latitude": -23.234234,
19    "longitude": -47.34567,
20    "localizacao": "LAB A103",
21    "responsavel": "Aluno - BELTRANO",
22    "unidade_medida": "unidade",
23    "status_operacional": true,
24    "observacao": "Sensor atualizado pela API no Insomnia"
25  }
26 ]
```

## 6 – Fazendo uma carga de dados de sensores

Um arquivo chamado **sensores.csv** foi montado com os dados levantados por todos em vários locais da escola.

Crie um arquivo com no nome **load\_sensors.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 os
import django
import csv
# Configure o ambiente do Django
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'smart_city.settings')
django.setup()

from app_smart.models import Sensor

def load_sensors_from_csv(file_path):
    with open(file_path, newline='', encoding='utf-8') as csvfile:
        reader = csv.DictReader(csvfile, delimiter=';')

        for row in reader:
            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
            )
    print('Dados carregados com sucesso!!!')

if __name__ == "__main__":
    # Caminho para o arquivo CSV
    file_path = 'dados/sensores.csv'
    load_sensors_from_csv(file_path)
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

Crie uma pasta com o nome **dados** no diretório raiz do seu projeto Django (o mesmo nível onde está o **manage.py**) e copie para esta pasta o arquivo **sensores.csv**.

Antes de fazer a carga desse arquivo no BD, vamos excluir os registros que inserimos como testes. Abra o DB Browser e execute o seguinte comando:

```
DELETE from app_smart_sensor where id > 0
```

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

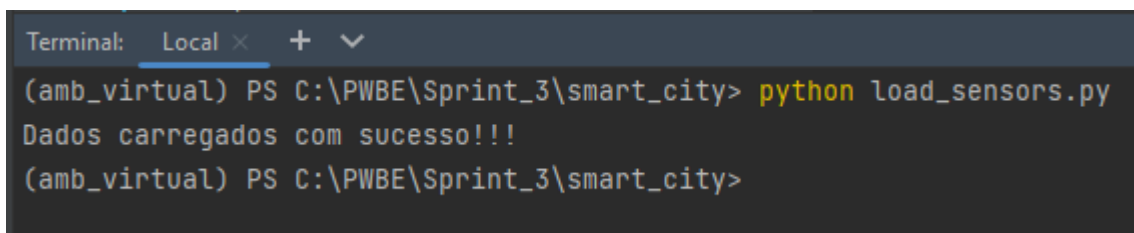
```
DELETE FROM sqlite_sequence WHERE name='app_smart_sensor';
```

Feche o BD Browser e retorne ao terminal.

Execute o comando como segue:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_sensors.py
```

Conforme pode ser observada no script, deve ser exibido a mensagem *'Dados carregados com sucesso!!!'*



```
Terminal: Local x + v
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_sensors.py
Dados carregados com sucesso!!!
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city>
```

Confira no BD se os dados foram carregados.

## 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:

```
from app_smart.models import Sensor, TemperaturaData
from rest_framework import permissions
from app_smart.api import serializers
from rest_framework.response import Response
from rest_framework import status
from rest_framework.views import APIView
from django.db.models import Q
```

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

```
class SensorFilterView(APIView):
    permission_classes = [permissions.IsAuthenticated]
    def post(self, request, *args, **kwargs):
        tipo = request.data.get('tipo', None)
        localizacao = request.data.get('localizacao', None)
        responsavel = request.data.get('responsavel', None)
        status_operacional = request.data.get('status operacional', None)
        filters = Q() # Inicializa um filtro vazio
        if tipo:
            filters &= Q(tipo__icontains=tipo)
        if localizacao:
            filters &= Q(localizacao__icontains=localizacao)
        if responsavel:
            filters &= Q(responsavel__icontains=responsavel)
        if status_operacional is not None:
            filters &= Q(status_operacional=status_operacional)
        queryset = Sensor.objects.filter(filters)
        serializer = serializers.SensorSerializer(queryset, many=True)
        return Response(serializer.data)
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

Em `urls.py` do `app_smart` acrescente a importação `SensorFilterView` que acabamos de criar:

A linha completa fica assim:

```
from app_smart.api.filters import (  
    SensorFilterView,  
)
```

Acrescente a seguinte rota na sessão `urlpatterns`

```
path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'),  
# Nova rota para filtragem personalizada
```

A sessão `urlpatterns` completa está assim agora:

```
urlpatterns = [  
    path('', views.abre_index, name='abre_index'),  
    path('api/create_user/', CreateUserAPIView.as_view(), name='create_user'),  
    path('usuarios', views.autenticacao, name='cad_user'),  
    path('cad_user', views.cad_user, name='cad_user'),  
    path('api/', include(router.urls)),  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),  
    path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'), # Nova rota para filtragem personalizada  
)
```

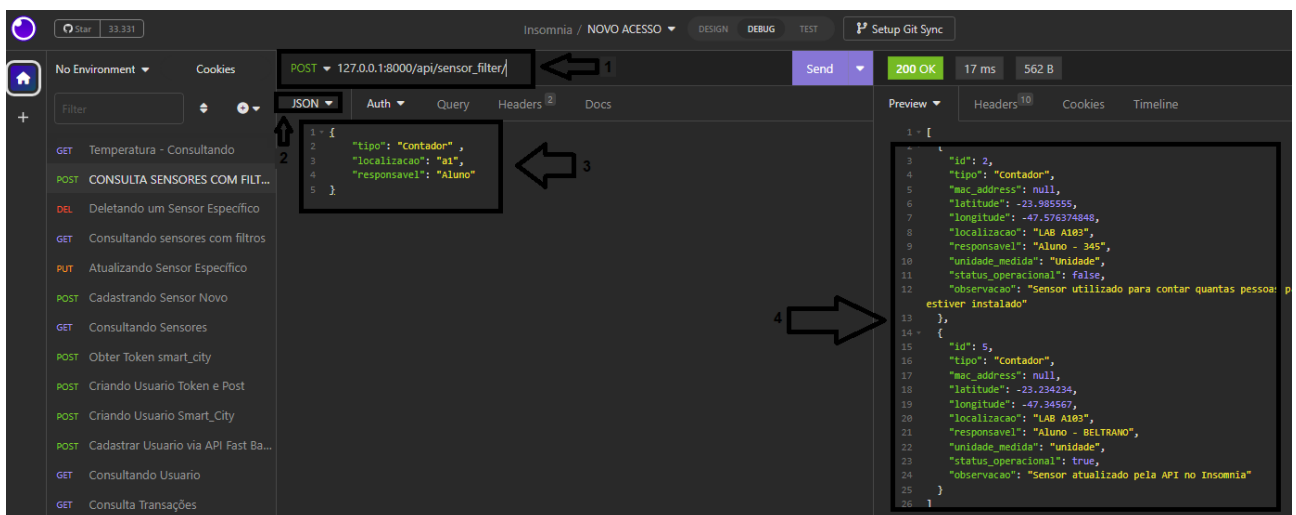
Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

## 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 API para 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.py*** 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 (
    CreateUserAPIView,
    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>

### 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 para 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.

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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.

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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:

**Execute o comando como segue:**

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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!!!’*

```
Terminal: Local x + v
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_sensors.py
Dados carregados com sucesso!!!
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city>
```

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 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 2024-04-12, 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** e acrescente 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)
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

```
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:

```
from app_smart.api.filters import (
    SensorFilterView,
    TemperaturaFilterView
)
```

Acrescente a seguinte rota na

sessão `urlpatterns`

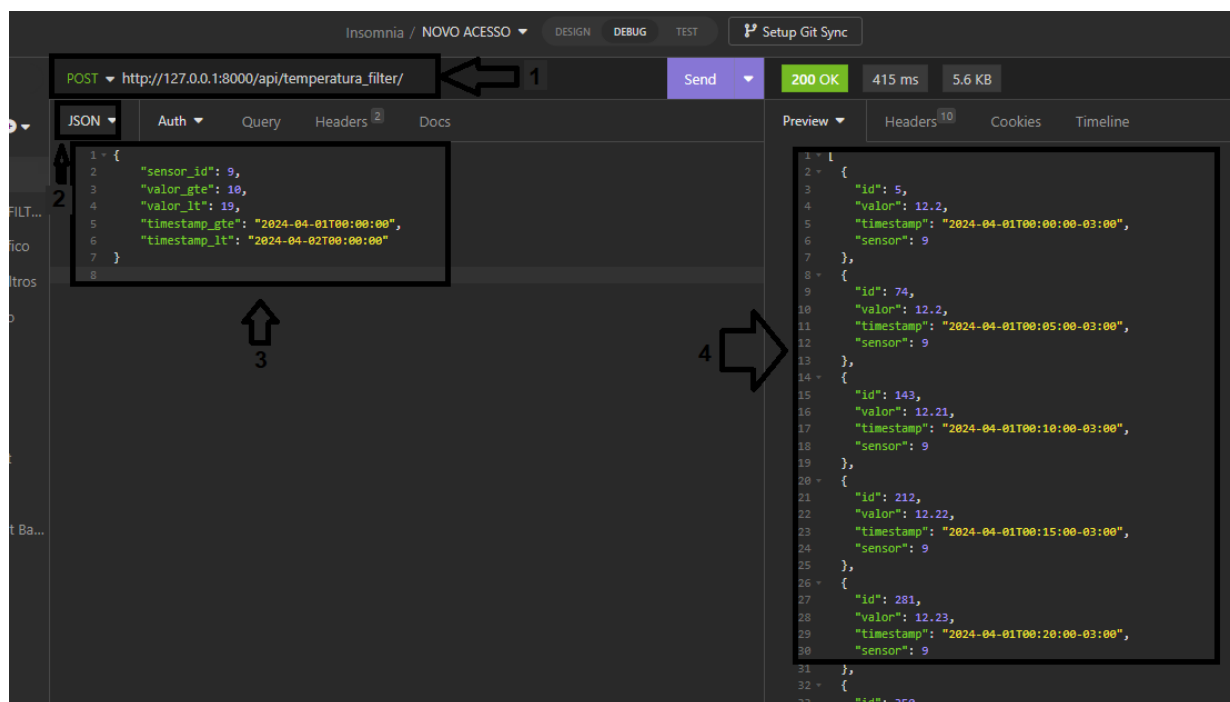
```
path('api/temperatura_filter/', TemperaturaFilterView.as_view(), name='temperatura_filter')
```

```
urlpatterns = [
    path('', views.abre_index, name='abre_index'),
    path('api/create_user/', CreateUserAPIViewSet.as_view(), name='create_user'),
    path('usuarios', views.autenticacao, name='cad_user'),
    path('cad_user', views.cad_user, name='cad_user'),
    path('api/', include(router.urls)),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'), # Nova rota para filtragem personalizada
    path('api/temperatura_filter/', TemperaturaFilterView.as_view(), name='sensor_filter'), # Nova rota para filtragem personalizada
]
```

### 9.6.1 – Testando a consulta da API `temperatura_filter` pelo Insomnia

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

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva



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 dos 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.py*** 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 *UmidadeDataSet*:

```
from app_smart.api.viewsets import (  
    CreateUserAPIViewSet,  
    SensorViewSet,  
    SensorFilterView,  
    TemperaturaDataSet,  
    UmidadeDataSet,  
)
```

Acrescente a linha:

```
router.register(r'umidade', UmidadeDataSet)
```

Com isso a nossa rota para a API que acabamos de criar vai ser <http://127.0.0.1:8000/api/umidade>

## 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_umidatedata where id > 0;
```

```
COMMIT;
```

```
DELETE FROM sqlite_sequence WHERE name='app_smart_umidatedata';
```

```
COMMIT
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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')
```

Execute o comando como segue:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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!!!'*

```
Terminal: Local x + v
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_sensors.py
Dados carregados com sucesso!!!
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city>
```

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
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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:

```
from app_smart.api.filters import (  
    SensorFilterView,  
    TemperaturaFilterView,  
    UmidadeFilterView  
)
```

Acrescente a seguinte rota na sessão `urlpatterns`  
`path('api/umidade_filter/', UmidadeFilterView.as_view(), name='umidade_filter'),`



Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

```
urlpatterns = [  
    path('', views.abre_index, name='abre_index'),  
    path('api/create_user/', CreateUserAPIView.as_view(), name='create_user'),  
    path('usuarios', views.autenticacao, name='cad_user'),  
    path('cad_user', views.cad_user, name='cad_user'),  
    path('api/', include(router.urls)),  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),  
    path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'), # Nova rota para filtrar  
    path('api/temperatura_filter/', TemperaturaFilterView.as_view(), name='temperatura_filter'), # Nova  
    path('api/umidade_filter/', UmidadeFilterView.as_view(), name='umidade_filter'), # Nova rota para filtrar
```

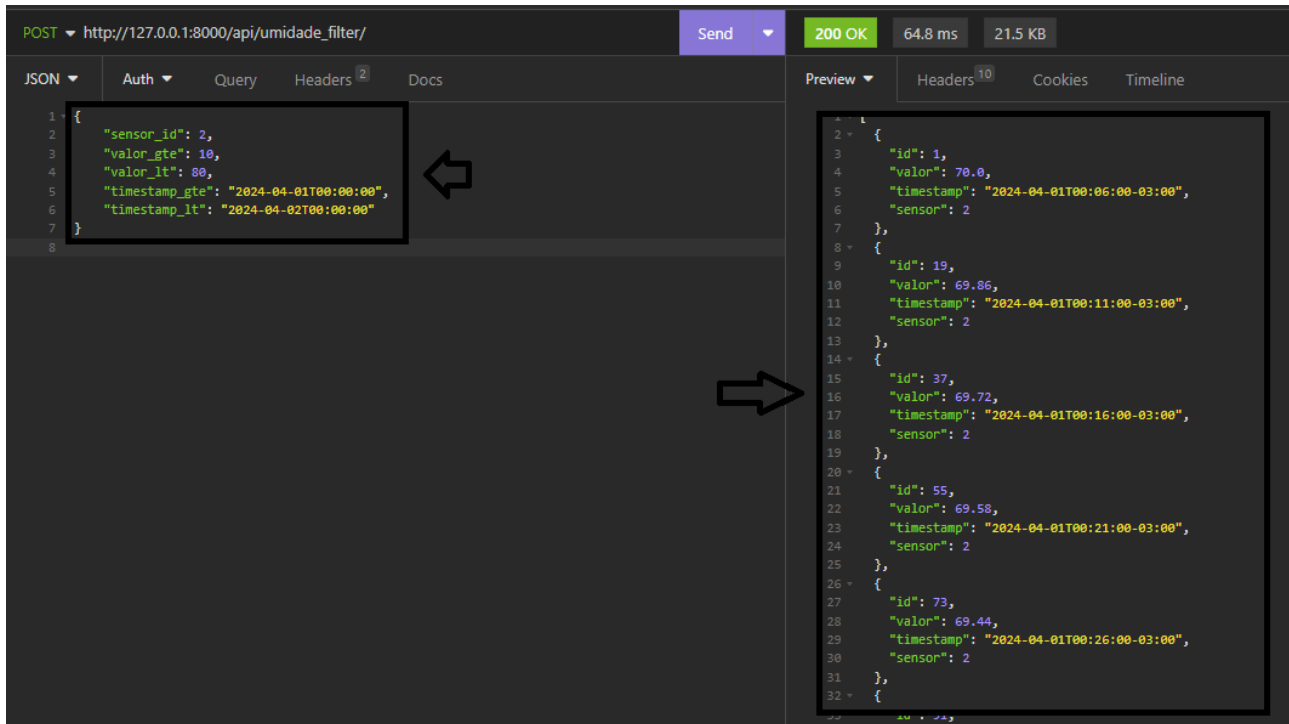
## 10.6 – Testando a consulta da API umidade\_filter pelo Insomnia.

**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.py** na pasta **api** em **smart city**.

## Acrescente a importação do model `LuminosidadeData`

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

```
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 (
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

```
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/luminosidade>

## 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
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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')
```

Execute o comando como segue:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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!!!’*

```
Terminal: Local x + v
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_sensors.py
Dados carregados com sucesso!!!
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city>
```

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
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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 :

```
from app_smart.api.filters import (
    SensorFilterView,
    TemperaturaFilterView,
    UmidadeFilterView,
    LuminosidadeFilterView
)
```

Acrescente a seguinte rota na sessão `urlpatterns`

```
path('api/luminosidade_filter/', LuminosidadeFilterView.as_view(),
     name='luminosidade_filter'),
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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

```
urlpatterns = [
    path('', views.abre_index, name='abre_index'),
    path('api/create_user/', CreateUserAPIViewSet.as_view(), name='create_user'),
    path('usuarios', views.autenticacao, name='cad_user'),
    path('cad_user', views.cad_user, name='cad_user'),
    path('api/', include(router.urls)),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'), # Nova rota para filtragem
    path('api/temperatura_filter/', TemperaturaFilterView.as_view(), name='temperatura_filter'), # Nova rota
    path('api/umidade_filter/', UmidadeFilterView.as_view(), name='umidade_filter'), # Nova rota para filtragem
    path('api/luminosidade_filter/', LuminosidadeFilterView.as_view(), name='luminosidade_filter'), # Nova rota
```

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

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

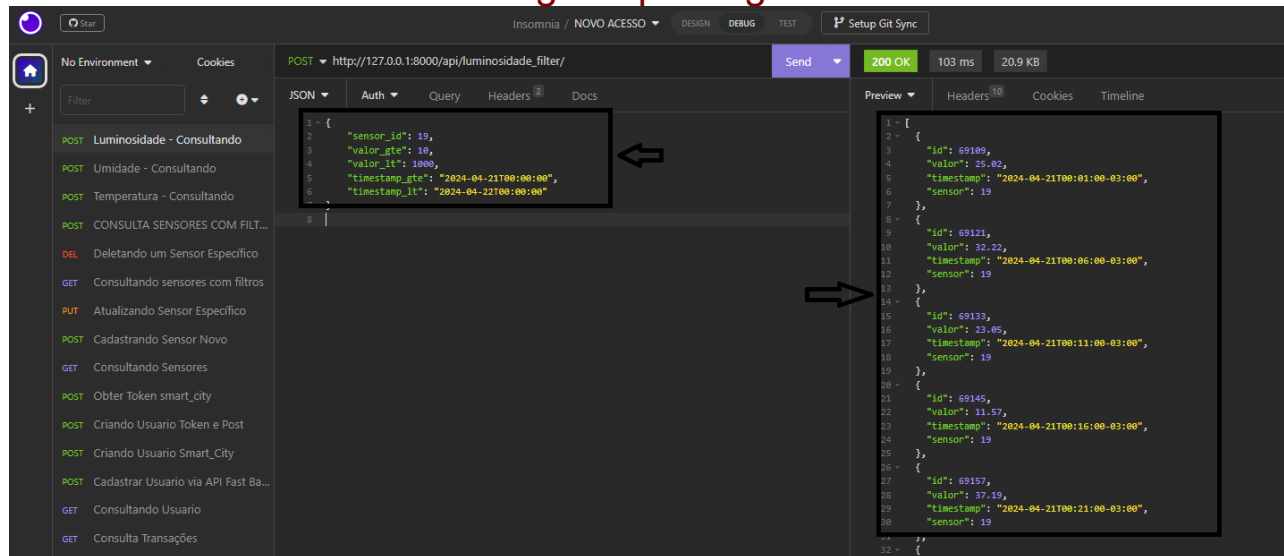
Utilize o seguinte JSON como exemplo para a busca:

```
{
  "sensor_id": 19,
  "valor_gte": 10,
  "valor_lt": 1000,
  "timestamp_gte": "2024-04-21T00:00:00",
  "timestamp_lt": "2024-04-22T00:00:00"
}
```



Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

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 `ContadorData` no arquivo `serializers.py`

Abra o mesmo arquivo **`serializers.py`** 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>

## 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.

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

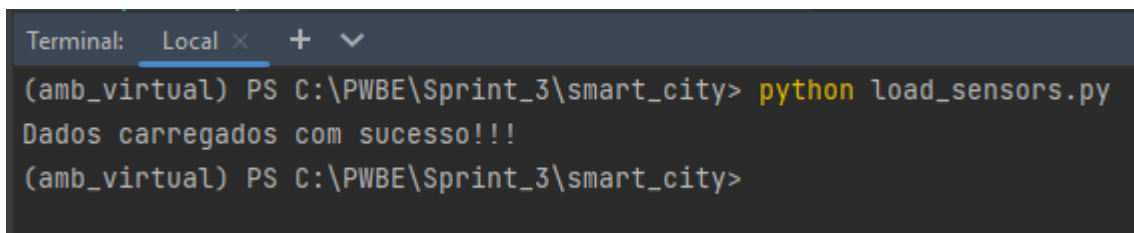
```
import csv
from datetime import datetime
from dateutil import parser
import pytz
import os
import django
# Configuração do Django
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'smart_city.settings')
django.setup()
from app_smart.models import ContadorData, Sensor
def load_contador_data(csv_file_path):
    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')
```

Execute o comando como segue:

```
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> 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!!!*



```
Terminal: Local x + v
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city> python load_contador.py
Dados carregados com sucesso!!!
(amb_virtual) PS C:\PWBE\Sprint_3\smart_city>
```

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, TemperaturaData, UmidadeData, LuminosidadeData, ContadorData
```

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:

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

A linha completa fica assim agora :

```
from app_smart.api.filters import (  
    SensorFilterView,  
    TemperaturaFilterView,  
    UmidadeFilterView,  
    LuminosidadeFilterView,  
    ContadorFilterView  
)
```

Acrescente a seguinte rota na sessão `urlpatterns`

```
path('api/contador_filter/', ContadorFilterView.as_view(),  
     name='contador_filter'),
```

A sessão *urlpatterns* deve ficar como segue:

```
urlpatterns = [  
    path('', views.abre_index, name='abre_index'),  
    path('api/create_user/', CreateUserAPIViewSet.as_view(), name='create_user'),  
    path('usuarios', views.autenticacao, name='cad_user'),  
    path('cad_user', views.cad_user, name='cad_user'),  
    path('api/', include(router.urls)),  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),  
    path('api/sensor_filter/', SensorFilterView.as_view(), name='sensor_filter'), # Nova rota  
    path('api/temperatura_filter/', TemperaturaFilterView.as_view(), name='temperatura_filter'),  
    path('api/umidade_filter/', UmidadeFilterView.as_view(), name='umidade_filter'), # Nova rota  
    path('api/luminosidade_filter/', LuminosidadeFilterView.as_view(), name='luminosidade_filter'),  
    path('api/contador_filter/', ContadorFilterView.as_view(), name='contador_filter'),  
]
```

Curso Análise e Desenvolvimento de Sistemas  
**Programação Web Back End**  
Material elaborado para o projeto Smart City – Prof. Israel Gomes da Silva

## 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:

The screenshot shows the Insomnia API client interface. On the left, a list of API endpoints is visible, including 'Contador', 'Luminosidade - Consultando', 'Umidade - Consultando', 'Temperatura - Consultando', 'CONSULTA SENSORES COM FILT...', 'Deletando um Sensor Especifico', 'Consultando sensores com filtros', 'Atualizando Sensor Especifico', 'Cadastrando Sensor Novo', 'Consultando Sensores', 'Obter Token smart\_city', 'Criando Usuario Token e Post', 'Criando Usuario Smart\_City', 'Cadastrar Usuario via API Fast Ba...', 'Consultando Usuario', and 'Consulta Transações'. The main panel displays a POST request to 'http://127.0.0.1:8000/api/contador\_filter/'. The request body is a JSON object: 

```
{  "sensor_id": 70,  "timestamp_gte": "2024-04-01T00:00:00",  "timestamp_lt": "2024-04-30T00:00:00"}
```

. The response is a 200 OK status with a response time of 130 ms and a body size of 14.9 KB. The response body is a JSON object: 

```
{  "count": 243,  "results": [    {      "id": 1,      "timestamp": "2024-04-01T09:21:00-03:00",      "sensor": 70    },    {      "id": 2,      "timestamp": "2024-04-01T09:26:00-03:00",      "sensor": 70    },    {      "id": 3,      "timestamp": "2024-04-01T09:31:00-03:00",      "sensor": 70    },    {      "id": 4,      "timestamp": "2024-04-01T14:51:00-03:00",      "sensor": 70    },    {      "id": 5,      "timestamp": "2024-04-01T14:56:00-03:00",      "sensor": 70    },    {      "id": 6,      "timestamp": "2024-04-01T15:01:00-03:00",      "sensor": 70    }  ]}
```