Material de apoio às aulas Django REST Framework e APIs RESTful

**Prof. José Alfredo Costa – Novembro 2024 – UFRN**

Veja o arquivo tutorial -> <https://drive.google.com/file/d/1gTWWmV6GcdVNpQIIZ_XOPLdiZGxc-_ND/view?usp=drive_link>

Veja o arquivo de apoio -> slides: <https://docs.google.com/document/d/15WRhsCYL3oZEmHWKBvzxPDDW4enpVtFszI92gqSXMM0/edit?tab=t.0>

Roteiro prático detalhado para cada uma das aulas

Inclui a sequência de atividades, exemplos de código, e orientações passo a passo para realizar cada tarefa com os alunos em sala de aula.

Sumário

[Aula 1: Introdução ao Django REST Framework e APIs RESTful 2](#_gjdgxs)

[**Objetivo Prático** 2](#_30j0zll)

[**Passo a Passo Prático** 2](#_1fob9te)

[Aula 2: Configuração do Ambiente e Instalação do DRF 4](#_3znysh7)

[**Objetivo Prático** 4](#_2et92p0)

[**Passo a Passo Prático** 4](#_tyjcwt)

[Aula 3: Principais Componentes do DRF 5](#_3dy6vkm)

[**Objetivo Prático** 5](#_1t3h5sf)

[**Passo a Passo Prático** 5](#_4d34og8)

[Aula 4: Implementação de um CRUD Completo no DRF 5](#_2s8eyo1)

[**Objetivo Prático** 5](#_17dp8vu)

[**Passo a Passo Prático** 5](#_3rdcrjn)

[Aula 5: Autenticação e Autorização no DRF 6](#_26in1rg)

[**Objetivo Prático** 6](#_lnxbz9)

[**Passo a Passo Prático** 6](#_35nkun2)

[Aula 6: Paginação e Filtragem 7](#_1ksv4uv)

[**Objetivo Prático** 7](#_44sinio)

[**Passo a Passo Prático** 7](#_2jxsxqh)

[Aula 7: Relacionamentos e Serializers Aninhados 8](#_z337ya)

[**Objetivo Prático** 8](#_3j2qqm3)

[**Passo a Passo Prático** 8](#_1y810tw)

[Aula 8: Segurança Avançada e Privacidade 10](#_4i7ojhp)

[**Objetivo Prático** 10](#_2xcytpi)

[**Passo a Passo Prático** 10](#_1ci93xb)

[Aula 9: Testes Automatizados e Documentação 11](#_3whwml4)

[**Objetivo Prático** 11](#_2bn6wsx)

[**Passo a Passo Prático** 12](#_qsh70q)

# Aula 1: Introdução ao Django REST Framework e APIs RESTful

**Objetivo Prático**

Familiarizar os alunos com o Django REST Framework e o conceito de APIs RESTful, com foco em criar uma estrutura básica de projeto.

**Passo a Passo Prático**

1. **Instalação do Django e DRF**
   * No terminal, oriente os alunos a instalar o Django e o DRF:

pip install django djangorestframework

* + Verifique se a instalação foi bem-sucedida:

python -m django --version

1. **Criando um Projeto Django**
   * No terminal:

django-admin startproject clinic\_api

cd clinic\_api

1. **Criando um App para Gerenciar Consultas**
   * No terminal:

python manage.py startapp consultas

1. **Configuração do DRF no Projeto**
   * No arquivo settings.py, adicione 'rest\_framework' em INSTALLED\_APPS.
2. **Criando o Modelo Consulta**
   * Em consultas/models.py:

python

from django.db import models

class Consulta(models.Model):

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

data = models.DateField()

horario = models.TimeField()

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

def \_\_str\_\_(self):

return f"{self.paciente} - {self.data} {self.horario}"

1. **Serializador para o Modelo Consulta**
   * Em consultas/serializers.py, defina o serializador:

python

from rest\_framework import serializers

from .models import Consulta

class ConsultaSerializer(serializers.ModelSerializer):

class Meta:

model = Consulta

fields = '\_\_all\_\_'

1. **Criação da APIView para Consultas**
   * Em consultas/views.py:

python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from .models import Consulta

from .serializers import ConsultaSerializer

class ConsultaListView(APIView):

def get(self, request):

consultas = Consulta.objects.all()

serializer = ConsultaSerializer(consultas, many=True)

return Response(serializer.data)

1. **Configurando URLs para a API**
   * Em consultas/urls.py:

python

from django.urls import path

from .views import ConsultaListView

urlpatterns = [

path('consultas/', ConsultaListView.as\_view(), name='consulta-list'),

]

* + Inclua essa URL em clinic\_api/urls.py:

python

from django.urls import path, include

urlpatterns = [

path('api/', include('consultas.urls')),

]

1. **Teste a API**
   * Execute o servidor:

python manage.py runserver

* + Acesse http://localhost:8000/api/consultas/ para ver a lista de consultas (vazia inicialmente).

# Aula 2: Configuração do Ambiente e Instalação do DRF

**Objetivo Prático**

Ensinar a configurar o ambiente de desenvolvimento com Django e DRF, e a criar um endpoint inicial para listar dados.

**Passo a Passo Prático**

1. **Instalação do Ambiente Virtual**
   * Crie e ative um ambiente virtual:

python -m venv env

source env/bin/activate

1. **Instalação das Dependências**
   * No ambiente virtual:

pip install django djangorestframework

1. **Configuração do Projeto e App**
   * Siga os passos da Aula 1 para configurar clinic\_api e o app consultas.
2. **Configuração do Modelo Consulta**
   * Siga o modelo da Aula 1 para o modelo Consulta.
3. **Configuração da View e Serializer**
   * Siga o modelo da Aula 1 para a ConsultaListView e ConsultaSerializer.
4. **Teste no Navegador**
   * Execute o servidor e acesse http://localhost:8000/api/consultas/.

# Aula 3: Principais Componentes do DRF

**Objetivo Prático**

Explorar os principais componentes: **ViewSets**, **Serializers** e **Routers** para criar uma API mais organizada e com menos código repetitivo.

**Passo a Passo Prático**

1. **Configuração do ViewSet**
   * Em consultas/views.py, crie ConsultaViewSet:

python

from rest\_framework import viewsets

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

1. **Configuração do Router**
   * Em consultas/urls.py:

python

from rest\_framework.routers import DefaultRouter

from .views import ConsultaViewSet

router = DefaultRouter()

router.register(r'consultas', ConsultaViewSet)

urlpatterns = router.urls

1. **Teste a API Completa no Navegador**
   * Execute o servidor e acesse http://localhost:8000/api/consultas/ para testar CRUD completo.

# Aula 4: Implementação de um CRUD Completo no DRF

**Objetivo Prático**

Implementar todas as operações CRUD para o modelo Paciente, usando Serializers, ViewSets e Routers.

**Passo a Passo Prático**

1. **Criação do Modelo Paciente**
   * Em consultas/models.py:

python

class Paciente(models.Model):

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

data\_nascimento = models.DateField()

telefone = models.CharField(max\_length=15)

email = models.EmailField(unique=True)

1. **Serializer do Paciente**
   * Em consultas/serializers.py:

python

class PacienteSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

fields = '\_\_all\_\_'

1. **ViewSet do Paciente**
   * Em consultas/views.py:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

1. **Configuração do Router**
   * Em consultas/urls.py:

python

router.register(r'pacientes', PacienteViewSet)

1. **Teste CRUD Completo**
   * Acesse http://localhost:8000/api/pacientes/.

# Aula 5: Autenticação e Autorização no DRF

**Objetivo Prático**

Configurar autenticação e controle de permissões na API.

**Passo a Passo Prático**

1. **Instalação do Token Authentication**
   * Adicione rest\_framework.authtoken em INSTALLED\_APPS e execute:

python manage.py migrate

1. **Configurando Token Authentication**
   * Em settings.py, configure:

python

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.TokenAuthentication',

],

1. **Gerando Tokens para Usuários**
   * Gere tokens no terminal:

python

python manage.py drf\_create\_token <username>

1. **Testar Token no Postman**
   * Teste enviando o token no cabeçalho Authorization: Token <seu\_token>.

# Aula 6: Paginação e Filtragem

**Objetivo Prático**

Configurar paginação e filtros para melhorar a usabilidade da API com grandes conjuntos de dados.

**Passo a Passo Prático**

1. **Configuração de Paginação**
   * Em settings.py, adicione PageNumberPagination.
2. **Configurando Filtros com django-filter**
   * Instale e configure django-filter no PacienteViewSet para filtrar nome e data\_nascimento.
3. **Testando Filtros e Paginação**
   * Execute a API e teste parâmetros de paginação e filtro na URL, como ?page=2 e ?nome=João.

Esses roteiros práticos fornecem aos alunos uma abordagem passo a passo e orientada para configurar e explorar o Django REST Framework, com foco na criação de uma API completa e segura.

# Aula 7: Relacionamentos e Serializers Aninhados

**Objetivo Prático**

Ensinar a configurar relacionamentos complexos no Django REST Framework (DRF) usando **serializers aninhados**. A aula utiliza exemplos com modelos de Paciente, Medico e Consulta, ilustrando os tipos de relacionamentos: **One-to-One**, **One-to-Many** e **Many-to-Many**.

**Passo a Passo Prático**

**Passo 1: Configuração dos Modelos Relacionados**

1. **Criar Modelo Paciente**:
   * Em consultas/models.py:

python

class Paciente(models.Model):

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

data\_nascimento = models.DateField()

telefone = models.CharField(max\_length=15)

email = models.EmailField(unique=True)

1. **Criar Modelo Medico**:
   * Em consultas/models.py:

python

class Medico(models.Model):

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

especialidade = models.CharField(max\_length=50)

crm = models.CharField(max\_length=20, unique=True)

1. **Criar Modelo Consulta com Relacionamentos**:
   * Adicione relações **Many-to-One** entre Consulta e Paciente e entre Consulta e Medico.
   * Em consultas/models.py:

python

class Consulta(models.Model):

paciente = models.ForeignKey(Paciente, on\_delete=models.CASCADE, related\_name='consultas')

medico = models.ForeignKey(Medico, on\_delete=models.CASCADE, related\_name='consultas')

data = models.DateField()

horario = models.TimeField()

**Passo 2: Criar Serializers Aninhados para os Relacionamentos**

1. **Serializer para Paciente**:
   * Em consultas/serializers.py:

python

class PacienteSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

fields = '\_\_all\_\_'

1. **Serializer para Medico**:
   * Em consultas/serializers.py:

python

class MedicoSerializer(serializers.ModelSerializer):

class Meta:

model = Medico

fields = '\_\_all\_\_'

1. **Serializer Aninhado para Consulta**:
   * No serializer ConsultaSerializer, inclua os serializers de Paciente e Medico.
   * Em consultas/serializers.py:

python

class ConsultaSerializer(serializers.ModelSerializer):

paciente = PacienteSerializer() # Aninhando serializer de Paciente

medico = MedicoSerializer() # Aninhando serializer de Medico

class Meta:

model = Consulta

fields = ['id', 'paciente', 'medico', 'data', 'horario']

**Passo 3: Configurando ViewSets e URLs**

1. **Criar ViewSets** para cada modelo em consultas/views.py:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

class MedicoViewSet(viewsets.ModelViewSet):

queryset = Medico.objects.all()

serializer\_class = MedicoSerializer

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

1. **Configuração de URLs**:
   * Em consultas/urls.py:

python

from rest\_framework.routers import DefaultRouter

from .views import PacienteViewSet, MedicoViewSet, ConsultaViewSet

router = DefaultRouter()

router.register(r'pacientes', PacienteViewSet)

router.register(r'medicos', MedicoViewSet)

router.register(r'consultas', ConsultaViewSet)

urlpatterns = router.urls

**Passo 4: Testando Serializers Aninhados**

* Execute o servidor e teste o endpoint /api/consultas/ para verificar que os dados do paciente e medico estão aninhados dentro de consulta.

**Exercício Final**

1. **Desafio**:
   * Altere ConsultaSerializer para permitir a criação de consultas enviando os IDs de paciente e medico sem serialização completa.

# Aula 8: Segurança Avançada e Privacidade

**Objetivo Prático**

Fortalecer a segurança da API implementando práticas como **autenticação baseada em grupos** e **ocultação de dados sensíveis**, além de configurar permissões personalizadas.

**Passo a Passo Prático**

**Passo 1: Configurando Grupos e Permissões no Django Admin**

1. **Criação de Grupos e Permissões**:
   * No Django Admin, crie grupos (Medico, Recepcionista, Administrador) e atribua permissões para cada grupo.
   * Associar permissões:
     + **Administrador**: Acesso total.
     + **Medico**: Criação e visualização de consultas.
     + **Recepcionista**: Somente visualização.

**Passo 2: Implementando Permissão Personalizada Baseada em Grupos**

1. **Permissão Baseada em Grupos**:
   * Em consultas/permissions.py:

python

from rest\_framework.permissions import BasePermission

class IsMedicoOrAdmin(BasePermission):

def has\_permission(self, request, view):

return request.user and (request.user.groups.filter(name='Medico').exists() or request.user.is\_staff)

1. **Aplicando a Permissão no ViewSet de Consultas**:
   * Em consultas/views.py:

python

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

permission\_classes = [IsMedicoOrAdmin]

**Passo 3: Configurando Serializers para Ocultar Dados Sensíveis**

1. **Serializers Condicionais para Ocultação de Dados**:
   * Em consultas/serializers.py:

python

class PacientePublicoSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

exclude = ['cpf']

class PacientePrivadoSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

fields = '\_\_all\_\_'

1. **Aplicando o Serializer no ViewSet de Paciente**:
   * Em consultas/views.py:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

def get\_serializer\_class(self):

if self.request.user.groups.filter(name='Medico').exists() or self.request.user.is\_staff:

return PacientePrivadoSerializer

return PacientePublicoSerializer

# Aula 9: Testes Automatizados e Documentação

**Objetivo Prático**

Implementar testes automatizados para garantir a qualidade da API e gerar documentação automatizada usando Swagger.

**Passo a Passo Prático**

**Passo 1: Configuração do Ambiente de Testes**

1. **Instalação do pytest e pytest-django**:

pip install pytest pytest-django

1. **Configuração do pytest**:
   * No diretório do projeto, crie pytest.ini:

ini

[pytest]

DJANGO\_SETTINGS\_MODULE = clinic\_api.settings

python\_files = tests.py test\_\*.py \*\_tests.py

**Passo 2: Criando Testes Unitários com unittest**

1. **Teste Unitário para Consulta**:
   * Em consultas/tests/test\_models.py:

python

from django.test import TestCase

from .models import Consulta, Paciente, Medico

class ConsultaModelTest(TestCase):

def setUp(self):

self.paciente = Paciente.objects.create(nome="João")

self.medico = Medico.objects.create(nome="Dra. Maria", crm="12345")

self.consulta = Consulta.objects.create(

paciente=self.paciente, medico=self.medico, data="2023-11-01", horario="10:00:00"

)

def test\_consulta\_criada(self):

self.assertEqual(self.consulta.paciente.nome, "João")

self.assertEqual(self.consulta.medico.nome, "Dra. Maria")

1. **Executar Testes**:

pytest

**Passo 3: Documentação da API com Swagger**

1. **Instalação do DRF-YASG**:

pip install drf-yasg

1. **Configuração das URLs para Documentação**:
   * Em clinic\_api/urls.py:

python

from drf\_yasg.views import get\_schema\_view

from drf\_yasg import openapi

from django.urls import path, include

schema\_view = get\_schema\_view(

openapi.Info(

title="API de Consultas Médicas",

default\_version="v1",

description="Documentação da API",

contact=openapi.Contact(email="contato@clinicamedica.com"),

license=openapi.License(name="MIT License"),

),

public=True,

)

urlpatterns = [

path('api/', include('consultas.urls')),

path('swagger/', schema\_view.with\_ui('swagger', cache\_timeout=0), name='schema-swagger-ui'),

path('redoc/', schema\_view.with\_ui('redoc', cache\_timeout=0), name='schema-redoc'),

]

1. **Testando Documentação**:
   * Acesse http://localhost:8000/swagger/ para visualizar e testar a documentação da API.

Esses roteiros detalhados para as aulas 7 a 9 fornecem uma abordagem prática para guiar os alunos no desenvolvimento de uma API robusta e segura com Django REST Framework. As etapas incluem a configuração de relacionamentos complexos, segurança avançada, testes automatizados e documentação, garantindo a qualidade e a acessibilidade da API.