**Guia 1**

Esquemas para slides de Aulas - Django REST Framework (DRF)

Prof. José Alfredo Costa – Novembro 2024 - UFRN

Dia 13/11

Lista:

Aula 1: Anna

Aula 2: Guilherme

Aula 3: Pedro

Aula 4: Jordan

Aula 5: Pablo / Leonardo

Aula 6: Victor

Aula 7: Mateus

Aula 8: Luiz / Aldo

Aula 9: João Rafael / Ricardo e Ivan

Aula 10: Ricardo e Ivan

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

Veja o arquivo de apoio -> Roteiro prático detalhado para cada uma das aulas: <https://docs.google.com/document/d/10PqvvYl2WgYSHjG_Sexqrzo6P8OaW0Xj_epImTKc2qo/edit?tab=t.0>

1. **Aula 1: Introdução ao Django REST Framework e APIs RESTful**
   * **Conteúdo**: Introdução ao DRF, conceito de APIs RESTful e princípios (recursos, métodos HTTP, diferenças entre APIs REST, SOAP e GraphQL).
   * **Objetivo**: Fornecer uma visão geral e entender a importância de APIs RESTful em aplicações modernas.
   * **Slides**: Conceitos básicos, estrutura REST, exemplos de aplicações de API, vantagens de usar REST.
   * **Prática**: Exercícios para criar uma visão simples de API REST.
2. **Aula 2: Configuração do Ambiente e Instalação do DRF**
   * **Conteúdo**: Preparação do ambiente, instalação do Django e DRF, configuração inicial.
   * **Objetivo**: Ensinar a configurar o DRF em um projeto Django do zero.
   * **Slides**: Passo a passo da instalação e configuração com capturas de tela e notas sobre cada configuração.
   * **Prática**: Criar o primeiro projeto Django com DRF, demonstrando o funcionamento básico.
3. **Aula 3: Principais Componentes do DRF (Views, Serializers, Routers)**
   * **Conteúdo**: Introdução e uso de Views, ViewSets, Serializers e Routers no DRF.
   * **Objetivo**: Ensinar a estrutura e uso de componentes essenciais para criação de APIs.
   * **Slides**: Descrição de cada componente, diagramas ilustrando a conexão entre eles, e exemplos.
   * **Prática**: Criar uma API simples com esses componentes, explorando as funcionalidades e visualizando o resultado no navegador.
4. **Aula 4: Implementação de um CRUD Completo no DRF**
   * **Conteúdo**: Criação de uma API com operações CRUD (Create, Read, Update, Delete) utilizando Serializers e ViewSets.
   * **Objetivo**: Fornecer uma aplicação prática completa de CRUD, importante para entender o ciclo de vida de dados em uma API.
   * **Slides**: Código passo a passo, explicando cada operação e como testá-la.
   * **Prática**: Implementação de um CRUD completo com um modelo simples (ex: Paciente).
5. **Aula 5: Autenticação e Autorização**
   * **Conteúdo**: Métodos de autenticação (Token, Session, JWT), configuração de permissões.
   * **Objetivo**: Ensinar a importância e implementação de segurança nas APIs, configurando autenticação e autorização.
   * **Slides**: Teoria sobre segurança, descrição dos métodos de autenticação e como configurar cada um.
   * **Prática**: Adicionar autenticação JWT e definir permissões de acesso para endpoints.
6. **Aula 6: Paginação e Filtragem de Dados**
   * **Conteúdo**: Implementação de paginação e filtragem, uso de django-filter.
   * **Objetivo**: Ajudar a otimizar a API para grandes conjuntos de dados e melhorar a experiência do usuário.
   * **Slides**: Explicação sobre tipos de paginação, exemplos de uso e código para configurar filtros.
   * **Prática**: Configurar paginação e filtros básicos em uma API.
7. **Aula 7: Relacionamentos e Serializers Aninhados**
   * **Conteúdo**: Serializers aninhados, tipos de relacionamento (One-to-One, One-to-Many, Many-to-Many).
   * **Objetivo**: Ensinar como lidar com relacionamentos complexos no DRF.
   * **Slides**: Diagramas de relacionamentos, exemplos de código e explicação sobre a necessidade de serializers aninhados.
   * **Prática**: Criar Serializers aninhados com modelos relacionados, simulando uma estrutura de consultas médicas com médicos e pacientes.
8. **Aula 8: Segurança Avançada e Privacidade**
   * **Conteúdo**: Proteção contra CSRF, autenticação baseada em grupos, ocultação de dados sensíveis.
   * **Objetivo**: Fortalecer as práticas de segurança e atender requisitos de privacidade.
   * **Slides**: Conceitos de segurança, explicações sobre proteção de dados e exemplos de configuração.
   * **Prática**: Implementar uma permissão personalizada e configurar controles de acesso mais detalhados.
9. **Aula 9: Testes Automatizados e Documentação**
   * **Conteúdo**: Testes automatizados (pytest e unittest), documentação da API (Swagger e DRF-YASG).
   * **Objetivo**: Ensinar como garantir a qualidade e a acessibilidade da API para outros desenvolvedores.
   * **Slides**: Explicação sobre tipos de testes, benefícios da documentação, exemplos de uso do Swagger.
   * **Prática**: Criar testes para uma API e gerar documentação com Swagger.

**Estrutura dos Slides**

1. **Título e Objetivo da Aula**: Sempre começar com um slide de introdução, indicando o objetivo da aula.
2. **Conteúdo Teórico**: Slides com conceitos básicos, com diagramas simples e esquemas ilustrando o funcionamento dos componentes.
3. **Exemplo de Código**: Slides dedicados a mostrar pequenos trechos de código, com explicações comentadas ao lado.
4. **Demonstrações e Práticas**: Slides que guiam a prática em passos, ajudando os alunos a seguirem as etapas em seus computadores.
5. **Desafios e Exercícios**: Slides com exercícios ao final de cada aula, incentivando os alunos a resolver problemas por conta própria.
6. **Recapitulação e Perguntas**: Encerrar com um resumo da aula e um slide para perguntas, promovendo uma revisão e discussão.

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

**Slide 1: Título e Objetivo da Aula**

* **Título**: Introdução ao Django REST Framework e APIs RESTful
* **Objetivo**: Entender o que é o Django REST Framework (DRF) e a importância de APIs RESTful no desenvolvimento de aplicações modernas.

**Slide 2: O que é Django REST Framework?**

* **Definição**: "DRF é uma biblioteca poderosa que facilita a criação de APIs em Django."
* **Pontos-Chave**:
  + Suporte a APIs RESTful.
  + Integração com o ORM do Django.
* **Imagem sugerida**: Diagrama básico mostrando o fluxo de dados entre a aplicação e o DRF, com clientes (aplicação móvel, web) consumindo a API.

**Slide 3: Por que Usar o Django REST Framework?**

* **Benefícios**:
  + Facilidade e rapidez no desenvolvimento de APIs.
  + Integração com autenticação e permissões do Django.
  + Documentação automática.
* **Exemplo Prático**: Expor dados de um sistema de gerenciamento de consultas médicas para que outros sistemas possam consumir.
* **Visual**: Ícones representando segurança, escalabilidade e flexibilidade.

**Slide 4: Conceitos Básicos de APIs RESTful**

* **O que é uma API?**
  + Conexão entre sistemas.
  + Permite comunicação entre aplicações.
* **Exemplo**: Uma API para uma clínica médica que retorna informações de consultas e médicos.
* **Imagem sugerida**: Um diagrama simples em Mermaid mostrando o fluxo Cliente -> API -> Banco de Dados.

mermaid

graph LR

Cliente --> API[API da Clínica]

API --> BancoDeDados[Banco de Dados]

**Slide 5: Princípios de APIs RESTful**

* **Princípios REST**:
  + **Client-Server**: Separação entre cliente e servidor.
  + **Stateless**: Cada requisição é independente.
  + **Cacheable**: Respostas podem ser armazenadas em cache.
  + **Layered System**: Sistema em camadas.
  + **Uniform Interface**: Interface consistente.
* **Diagrama em Mermaid**:

mermaid

graph TD

Cliente -->|Request| Servidor

Servidor -->|Response| Cliente

Cliente -.->|Cache| Servidor

Cliente -->|Uniform Interface| Servidor

**Slide 6: Métodos HTTP e APIs RESTful**

* **Métodos HTTP**:
  + **GET**: Obter dados.
  + **POST**: Criar um recurso.
  + **PUT**: Atualizar um recurso.
  + **DELETE**: Remover um recurso.
* **Exemplo de Uso**: “Criar um endpoint para listar e agendar consultas médicas.”
* **Tabela** com métodos e exemplos de uso na clínica médica.

**Slide 7: Vantagens do RESTful**

* Comparação com **SOAP e GraphQL**.
* **REST**:
  + Mais leve e simples.
  + Suporte universal.
* **SOAP**:
  + Estrutura mais rígida e complexa.
* **Imagem sugerida**: Diagrama de comparação com destaque para a simplicidade do REST.

**Slide 8: Exercício Prático - Criando uma API Simples**

* **Tarefa**: "Crie uma API para listar consultas médicas."
* **Dica**: Usar APIView e Response no DRF.
* **Código básico** para iniciar a API.

# Aula 2: Configuração do Ambiente e Instalação do DRF

**Slide 1: Título e Objetivo da Aula**

* **Título**: Configuração do Ambiente e Instalação do DRF
* **Objetivo**: Configurar o ambiente de desenvolvimento, instalar o Django e o DRF, e criar um projeto inicial.

**Slide 2: Preparação do Ambiente**

* **Instalando o Python**:
  + Comando para verificar instalação: python --version.
  + Link para download: [Python.org](https://www.python.org/downloads/).
* **Imagem sugerida**: Captura de tela do site do Python e do terminal.

**Slide 3: Instalando o Django**

* **Comando**: pip install django.
* **Criando um Projeto Django**:
  + Comando para iniciar o projeto: django-admin startproject clinic\_api.
* **Diagrama em Mermaid**: Mostrando a sequência da instalação e criação do projeto.

mermaid

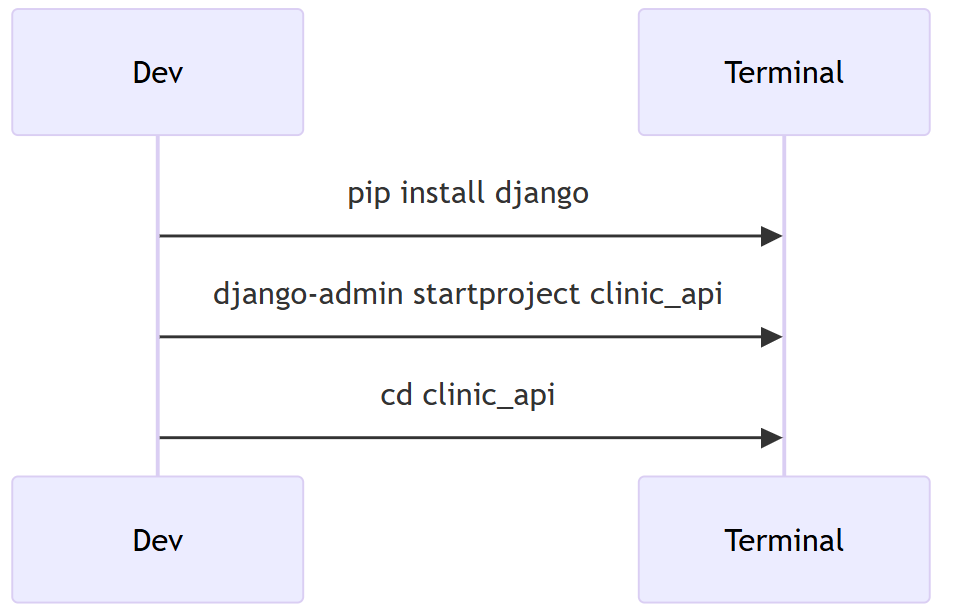
sequenceDiagram

participant Dev

Dev->>Terminal: pip install django

Dev->>Terminal: django-admin startproject clinic\_api

Dev->>Terminal: cd clinic\_api

[](https://mermaid.live/view#pako:eNptkMFugzAMhl8l8pkiIEAgh5563Wk7TUiTlaRtKkjSkEzrEO--rKjSJtUn__Zny_4XEFYq4DCra1RGqIPGk8dpMCSFQx-00A5NIAf1uRVTstvv35SftMGRE6cd0WYOOI5EXtCc7HNu6-1QJk0S7oPz9qJEIGLURosPdPr5pJD_EMhgSj3UMt29_I4MEM5qUgPwlEp1xDiGAQazJhRjsK83I4AHH1UG3sbTGfgRxzmp6CSGx9MPJP37bu1fCXyBL-AlLfOqKRitqrav-6phGdyAd3VelFXHWNv3JWtou2bwfV9Q5F1N65axkjZ9WbUtzUBJHax_2Wy_u7_-AJR5fYM)

Veja em: <https://mermaid.live/edit#pako:eNptkU1rwzAMhv-K0TkJidMutg899brTdhqBIWw3dUlsz7HHupD_Pq-hsEF10is9EvpYQDqlQcCsP5K2Uh8NDgGn3pJsHkM00ni0kRz15xbMTnk4vOowGYujIN54YuwccRyJuqAd3GNuy5WosiYZD9EHd9EyEjkaa-Q7evO4Uqp_CBQw5Rwaledefkt6iGc96R5EdpU-YRpjD71dM4opuperlSBiSLqA4NJwBnHCcc4qeYXxvvQdyfu-OfdXgljgC0TZVN2O04Zz1vJ6z-m-K-Ca47TirGta2vEdpZzSp3Yt4PvWo64yzTrW1nXDOO8aVoBWJrrwvF3-9oD1BwYRfjI>

**Slide 4: Instalando o Django REST Framework**

* **Comando**: pip install djangorestframework.
* **Configuração** no settings.py para adicionar o DRF.
* **Imagem sugerida**: Captura de tela mostrando a edição do arquivo settings.py.

**Slide 5: Configuração Básica do DRF**

* **Arquivo**: settings.py.
  + INSTALLED\_APPS: Adicionar 'rest\_framework'.
  + REST\_FRAMEWORK: Configurações iniciais de permissão e renderização.
* **Exemplo de Código**:

python

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

...

'rest\_framework',

]

REST\_FRAMEWORK = {

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.AllowAny',

],

'DEFAULT\_RENDERER\_CLASSES': [

'rest\_framework.renderers.JSONRenderer',

],

}

**Slide 6: Criando um Aplicativo no Django**

* **Comando**: python manage.py startapp consultas.
* **Configuração**: Adicionar o app consultas ao INSTALLED\_APPS.
* **Imagem**: Exemplo do arquivo settings.py com o app adicionado.

**Slide 7: Configurando URLs e Primeira View**

* **Criar arquivo urls.py** no app consultas.
* **Adicionar URL no projeto principal**: path('api/', include('consultas.urls')).
* **Exemplo de Código**:

python

from django.urls import path

from .views import ListaConsultasView

urlpatterns = [

path('consultas/', ListaConsultasView.as\_view(), name='lista\_consultas'),

]

**Slide 8: Testando a Configuração**

* **Iniciar o servidor**: python manage.py runserver.
* **Acessar**: http://localhost:8000/api/consultas/.
* **Imagem sugerida**: Exemplo de resposta JSON para uma requisição de lista de consultas.

**Slide 9: Exercício Prático - Implementação Básica**

* **Tarefa**: "Configure o ambiente e crie uma view simples para listar as consultas médicas."
* **Dica**: Usar os passos anteriores como referência.
* **Desafio**: Adicionar um modelo básico e listar os dados na API.

Com esses slides e materiais, você terá uma aula inicial bem estruturada, com teoria, prática e visuais para auxiliar o entendimento dos alunos.

A Aula 3 é fundamental para introduzir os principais componentes do Django REST Framework: **Views**, **ViewSets**, **Serializers** e **Routers**. Vou estruturar os slides com uma abordagem prática e detalhada, oferecendo exemplos de código, diagramas de fluxo e explicações passo a passo para os alunos.

# Aula 3: Principais Componentes do Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Principais Componentes do Django REST Framework
* **Objetivo**: Compreender o uso de Views, ViewSets, Serializers e Routers no DRF para criar uma API completa.

**Slide 2: Introdução aos Componentes do DRF**

* **DRF Components Overview**:
  + **Views e ViewSets**: Controlam o comportamento das requisições HTTP (GET, POST, PUT, DELETE).
  + **Serializers**: Transformam dados entre o formato JSON e o formato de modelos Django.
  + **Routers**: Simplificam o roteamento de URLs para endpoints da API.
* **Diagrama de Relação Entre os Componentes**:

mermaid

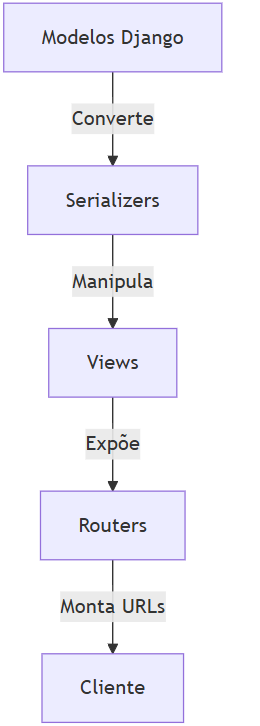
graph TD

Modelos[Modelos Django] -->|Converte| Serializers

Serializers -->|Manipula| Views

Views -->|Expõe| Routers

Routers -->|Monta URLs| Cliente

[](https://mermaid.live/view#pako:eNpNUMtywjAM_BWPzoEhzoPYh16gt3KBtoc2PWiIAHcSO-PYLY_wWf2C_ljdQDvoIq20uyPpBGtTEUjYWmx37HFeahZiEZq16V6vmc3fUW_NGxuN7vqZ0R9kHfVsRVZhrY5ku4vspjFQF6hV62vs2bOizytpKIfx_b79_go-S-Pdv8cVXPRGO2RPy4euZ7NakXZUaoigIdugqsLap19RCW5HDZUgQ1nRBn3tSij1OVDRO7M66DVIZz1FYI3f7kBusO4C8m2FjuYKw_nNH6VF_WLMLQR5gj3IdJqMs5yLaVrwNOdxFsEB5Cgdi2yaiGKS80kRJ4KfIzgOBpOxKJI05qLIipjneSYioEo5YxeXrw_PP_8AJzJ9UA)

**Slide 3: Views no Django REST Framework**

* **O que são Views?**
  + Classes que manipulam requisições HTTP.
  + Permitem definir operações específicas para cada método HTTP.
* **Tipos de Views**:
  + **APIView**: Classe básica do DRF para criar views personalizadas.
  + **Generic Views**: Views mais abstratas que facilitam o uso de padrões CRUD.
  + **ViewSets**: Combina múltiplas operações em uma única classe, ideal para CRUD.
* **Exemplo**: Criando uma View para listar consultas médicas.

**Slide 4: Exemplo de APIView no DRF**

* **Código para Criar uma APIView Básica**: ListaConsultasView para retornar uma lista de consultas.

python

from rest\_framework.views import APIView

from rest\_framework.response import Response

from .models import Consulta

from .serializers import ConsultaSerializer

class ListaConsultasView(APIView):

def get(self, request):

consultas = Consulta.objects.all()

serializer = ConsultaSerializer(consultas, many=True)

return Response(serializer.data)

* **Explicação do Código**:
  + APIView: Classe base para criar views no DRF.
  + get: Método HTTP para obter dados.
  + ConsultaSerializer: Serializador para transformar dados de consulta em JSON.

**Slide 5: ViewSets - Simplificando Operações CRUD**

* **O que são ViewSets?**
  + São classes que permitem agrupar várias operações CRUD.
  + Recomendado para APIs com estrutura padrão de CRUD.
* **Diagrama**: Demonstração da simplificação do CRUD com ViewSet.

mermaid

classDiagram

class ConsultaViewSet {

+list(request)

+create(request)

+retrieve(request, pk)

+update(request, pk)

+destroy(request, pk)

}

* **Benefícios**:
  + Menos código repetitivo.
  + Integração direta com Routers para geração de URLs.

**Slide 6: Exemplo de ViewSet no DRF**

* **Código para Criar um ViewSet Completo**: ConsultaViewSet para CRUD de consultas.

python

from rest\_framework import viewsets

from .models import Consulta

from .serializers import ConsultaSerializer

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

* **Explicação do Código**:
  + ModelViewSet: Uma classe de ViewSet que já inclui operações CRUD.
  + queryset: Define o conjunto de dados (todas as consultas).
  + serializer\_class: Serializador para manipular os dados.

**Slide 7: Serializers - Convertendo Dados para JSON**

* **O que são Serializers?**
  + Serializers convertem dados entre JSON e os modelos Django.
  + São usados para garantir que os dados estejam no formato correto.
* **Tipos de Serializers**:
  + **ModelSerializer**: Cria serializadores com base nos modelos Django.
  + **Serializers personalizados**: Para cenários onde o formato JSON precisa ser ajustado.
* **Diagrama**: Exemplo de serialização.

mermaid

graph TD

Model[Modelo Consulta] -->|Converte para JSON| Serializer

Serializer --> JSON[Formato JSON]

**Slide 8: Exemplo de Serializer para o Modelo Consulta**

* **Código para Criar um Serializer** para o modelo Consulta.

python

from rest\_framework import serializers

from .models import Consulta

class ConsultaSerializer(serializers.ModelSerializer):

class Meta:

model = Consulta

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

* **Explicação do Código**:
  + ModelSerializer: Simplifica a criação de Serializers com base em modelos.
  + fields = '\_\_all\_\_': Inclui todos os campos do modelo Consulta no JSON.

**Slide 9: Adicionando Serializers Aninhados (Opcional)**

* **Quando usar Serializers Aninhados?**
  + Quando é necessário incluir dados de modelos relacionados.
  + Exemplo: incluir os detalhes do médico junto com a consulta.
* **Código de Exemplo**:

python

class MedicoSerializer(serializers.ModelSerializer):

class Meta:

model = Medico

fields = ['nome', 'crm', 'especialidade']

class ConsultaDetalhadaSerializer(serializers.ModelSerializer):

medico = MedicoSerializer()

class Meta:

model = Consulta

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

**Slide 10: Routers - Mapeando URLs para ViewSets**

* **O que são Routers?**
  + Automatizam o mapeamento de URLs para ViewSets.
  + Simplificam o processo de criação de URLs para operações CRUD.
* **Diagrama**: Demonstração de mapeamento com Routers.

mermaid

flowchart LR

Router -->|Gera URLs| ViewSet

ViewSet -->|CRUD| URLs[URLs para API]

URLs --> Cliente

**Slide 11: Exemplo de Uso de Router**

* **Código para Configurar o Router e o ViewSet de Consulta**.

python

from rest\_framework.routers import DefaultRouter

from .views import ConsultaViewSet

from django.urls import path, include

router = DefaultRouter()

router.register(r'consultas', ConsultaViewSet)

urlpatterns = [

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

]

* **Explicação do Código**:
  + DefaultRouter: Classe que facilita a geração de URLs para APIs.
  + register: Associa o ViewSet ConsultaViewSet à URL /consultas/.

**Slide 12: Testando a API com Views, Serializers e Routers**

* **Passo a Passo para Testar a API**:
  1. Iniciar o servidor com python manage.py runserver.
  2. Acessar http://localhost:8000/api/consultas/.
  3. Utilizar uma ferramenta como Postman para enviar requisições de teste (GET, POST, PUT, DELETE).
* **Dicas**:
  + **GET**: Lista todas as consultas.
  + **POST**: Cria uma nova consulta (enviar JSON com dados).
  + **PUT**: Atualiza uma consulta existente (enviar JSON com dados atualizados).
  + **DELETE**: Remove uma consulta específica.

**Slide 13: Exercício Prático - Criando um CRUD Completo**

* **Tarefa**: "Crie um CRUD completo para o modelo Paciente, usando ViewSet, Serializer e Router."
* **Passos Sugeridos**:
  1. Criar o PacienteSerializer usando ModelSerializer.
  2. Configurar o PacienteViewSet com ModelViewSet.
  3. Adicionar o ViewSet ao DefaultRouter e incluir no urlpatterns.
* **Desafio**: Testar todos os métodos CRUD com uma ferramenta de requisição HTTP.

**Slide 14: Recapitulando a Aula**

* **Resumo dos Componentes**:
  + **Views**: Controlam o fluxo das requisições.
  + **ViewSets**: Simplificam operações CRUD.
  + **Serializers**: Transformam dados para JSON.
  + **Routers**: Geram automaticamente URLs.
* **Benefícios**: Estrutura organizada, menos código repetitivo, e facilidade de manutenção.

Esses slides cobrem a Aula 3 com bastante detalhe, incluindo passo a passo para configurar cada componente, exemplos de código, diagramas, e um exercício prático para aplicar o que foi aprendido.

A Aula 4 será focada na **Implementação de um CRUD Completo no DRF**. Nesta aula, os alunos aprenderão a implementar todas as operações CRUD (Create, Read, Update, Delete) para um modelo Django usando o DRF, com uma abordagem prática que inclui explicações detalhadas, diagramas e exemplos de código.

# Aula 4: Implementação de um CRUD Completo no Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Implementação de um CRUD Completo no Django REST Framework
* **Objetivo**: Aprender a criar e configurar um CRUD completo no DRF para o modelo Paciente, utilizando Serializers, ViewSets, Routers, e testando cada operação (Create, Read, Update, Delete).

**Slide 2: Estrutura de um CRUD Completo no DRF**

* **Operações CRUD**:
  + **Create**: Criar um novo recurso.
  + **Read**: Ler recursos existentes.
  + **Update**: Atualizar recursos existentes.
  + **Delete**: Remover recursos existentes.
* **Ferramentas**:
  + **Model**: Define o esquema dos dados (ex.: Paciente).
  + **Serializer**: Converte os dados entre JSON e o modelo Django.
  + **ViewSet**: Contém as operações CRUD.
  + **Router**: Configura as URLs automaticamente.
* **Diagrama de Relação CRUD**:

mermaid

graph TD

Model[Modelo Paciente] --> Serializer

Serializer --> ViewSet

ViewSet --> Router

Router --> API[Endereços da API]

API --> Cliente

**Slide 3: Preparando o Modelo Paciente**

* **Definição do Modelo Paciente**:
  + Vamos criar um modelo simples chamado Paciente com alguns campos relevantes.
* **Código do Modelo**:

python

from django.db import models

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)

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

return self.nome

* **Explicação do Código**:
  + nome: Nome completo do paciente.
  + data\_nascimento: Data de nascimento do paciente.
  + telefone: Número de telefone.
  + email: Endereço de e-mail único para identificação.

**Slide 4: Criando o Serializer para o Paciente**

* **Por que precisamos de um Serializer?**
  + Serializers convertem os dados do modelo para JSON, permitindo que eles sejam enviados e recebidos pela API.
* **Código para o Serializer**: PacienteSerializer.

python

from rest\_framework import serializers

from .models import Paciente

class PacienteSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

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

* **Explicação do Código**:
  + ModelSerializer: Simplifica a criação do Serializer com base no modelo.
  + fields = '\_\_all\_\_': Inclui todos os campos do modelo Paciente no JSON.

**Slide 5: Configurando o ViewSet para Paciente**

* **O que é um ViewSet?**
  + Um ViewSet permite agrupar todas as operações CRUD em uma única classe.
  + Usaremos o ModelViewSet, que já fornece métodos para criar, ler, atualizar e excluir dados.
* **Código para o ViewSet**: PacienteViewSet.

python

from rest\_framework import viewsets

from .models import Paciente

from .serializers import PacienteSerializer

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

* **Explicação do Código**:
  + queryset: Define o conjunto de dados que será usado para as operações.
  + serializer\_class: Define qual serializer será utilizado.

**Slide 6: Configurando o Router para o ViewSet**

* **O que é um Router?**
  + O Router mapeia automaticamente URLs para as operações do ViewSet.
* **Código para Configurar o Router e o ViewSet**:

python

from rest\_framework.routers import DefaultRouter

from .views import PacienteViewSet

from django.urls import path, include

router = DefaultRouter()

router.register(r'pacientes', PacienteViewSet)

urlpatterns = [

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

]

* **Explicação do Código**:
  + DefaultRouter: Gera URLs automaticamente para as operações CRUD.
  + router.register(r'pacientes', PacienteViewSet): Registra o ViewSet PacienteViewSet no endpoint /pacientes/.

**Slide 7: Testando as Operações CRUD no Browser e no Postman**

* **Iniciando o servidor**:
  + Comando: python manage.py runserver
  + URL Base: http://localhost:8000/api/pacientes/
* **Métodos CRUD**:
  + **GET**: http://localhost:8000/api/pacientes/ para listar todos os pacientes.
  + **POST**: Enviar JSON para criar um novo paciente.
  + **PUT**: Atualizar um paciente existente.
  + **DELETE**: Remover um paciente específico.
* **Imagem sugerida**: Tela de exemplo do Postman com uma requisição POST para criar um paciente.

**Slide 8: Testando o Método GET (Listar Pacientes)**

* **Descrição**: O método GET recupera todos os pacientes ou um paciente específico.
* **Exemplo de Código no ViewSet**:

python

def list(self, request):

queryset = Paciente.objects.all()

serializer = PacienteSerializer(queryset, many=True)

return Response(serializer.data)

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/
  + Resposta esperada (JSON):

json

[

{

"id": 1,

"nome": "João Silva",

"data\_nascimento": "1990-05-15",

"telefone": "123456789",

"email": "joao@example.com"

}

]

**Slide 9: Testando o Método POST (Criar Paciente)**

* **Descrição**: O método POST cria um novo paciente.
* **Exemplo de Código no ViewSet**:

python

def create(self, request):

serializer = PacienteSerializer(data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=201)

return Response(serializer.errors, status=400)

* **Exemplo de JSON para Criar um Paciente**:

json

{

"nome": "Maria Oliveira",

"data\_nascimento": "1985-10-22",

"telefone": "987654321",

"email": "maria@example.com"

}

* **Resposta esperada**: JSON do paciente criado.

**Slide 10: Testando o Método PUT (Atualizar Paciente)**

* **Descrição**: O método PUT atualiza um paciente existente.
* **Exemplo de Código no ViewSet**:

python

def update(self, request, pk=None):

paciente = self.get\_object()

serializer = PacienteSerializer(paciente, data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data)

return Response(serializer.errors, status=400)

* **Exemplo de JSON para Atualizar um Paciente**:

json

{

"nome": "Maria Oliveira Silva",

"data\_nascimento": "1985-10-22",

"telefone": "987654321",

"email": "maria.silva@example.com"

}

* **Resposta esperada**: JSON do paciente atualizado.

**Slide 11: Testando o Método DELETE (Remover Paciente)**

* **Descrição**: O método DELETE remove um paciente específico.
* **Exemplo de Código no ViewSet**:

python

def destroy(self, request, pk=None):

paciente = self.get\_object()

paciente.delete()

return Response(status=204)

* **Exemplo de Requisição DELETE**:
  + URL: http://localhost:8000/api/pacientes/1/
  + **Resposta esperada**: Status HTTP 204 (No Content).

**Slide 12: Exercício Prático - Criando CRUD Completo para outro Modelo**

* **Tarefa**: "Implemente um CRUD completo para o modelo Consulta, usando Serializers, ViewSet e Router."
* **Passos Sugeridos**:
  1. Criar o ConsultaSerializer.
  2. Configurar o ConsultaViewSet.
  3. Adicionar o ViewSet ao Router em urls.py.
* **Desafio**: Testar todas as operações CRUD com o modelo Consulta.

**Slide 13: Recapitulando a Aula**

* **Resumo dos Passos para Implementar um CRUD Completo**:
  1. Definir o modelo.
  2. Criar o Serializer para o modelo.
  3. Configurar o ViewSet com ModelViewSet.
  4. Registrar o ViewSet no Router para gerar URLs.
  5. Testar as operações CRUD com uma ferramenta como Postman.
* **Benefícios do CRUD com ViewSet**: Menos código, padrão DRY, e facilidade de manutenção.

Essa estrutura para a Aula 4 proporciona um guia detalhado para implementar e testar um CRUD completo no Django REST Framework, com explicações para cada etapa, exemplos de código e exercício prático.

A Aula 5 será dedicada aos conceitos de **Autenticação e Autorização** no Django REST Framework (DRF). Nessa aula, vamos explorar os diferentes métodos de autenticação oferecidos pelo DRF, como o uso de tokens e JWT (JSON Web Tokens), e como configurar permissões para limitar o acesso a certos recursos da API.

# Aula 5: Autenticação e Autorização no Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Autenticação e Autorização no Django REST Framework
* **Objetivo**: Aprender a implementar autenticação e controle de permissões em uma API com o DRF, utilizando métodos como Token e JWT.

**Slide 2: Introdução à Autenticação e Autorização**

* **Autenticação vs. Autorização**:
  + **Autenticação**: Verificação da identidade do usuário (ex.: login).
  + **Autorização**: Controle do acesso a recursos com base na identidade.
* **Por que são importantes?**:
  + Garantem segurança e protegem dados sensíveis.
  + Limitam o acesso a usuários com permissões específicas.

**Slide 3: Métodos de Autenticação no DRF**

* **Principais métodos oferecidos pelo DRF**:
  + **Token Authentication**: Autenticação baseada em tokens exclusivos.
  + **JWT (JSON Web Tokens)**: Autenticação com tokens gerados a partir de informações do usuário.
  + **Session Authentication**: Autenticação baseada em sessões (comumente usada para aplicações web).
* **Diagrama de Fluxo de Autenticação**:

mermaid

flowchart LR

Cliente -- Envia Credenciais --> Servidor

Servidor -- Retorna Token --> Cliente

Cliente -- Envia Token --> Servidor

Servidor -- Verifica Token --> Recurso[Recursos da API]

**Slide 4: Configuração Básica de Autenticação no DRF**

* **Passo 1**: Adicionar rest\_framework.authtoken no INSTALLED\_APPS.
* **Passo 2**: Configurar o DRF para usar autenticação de token.
* **Código no settings.py**:

python

INSTALLED\_APPS = [

...

'rest\_framework',

'rest\_framework.authtoken', # Necessário para autenticação por token

]

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.TokenAuthentication',

],

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.IsAuthenticated',

],

}

* **Explicação**:
  + TokenAuthentication: Ativa a autenticação via token para a API.
  + IsAuthenticated: Garante que apenas usuários autenticados possam acessar a API por padrão.

**Slide 5: Gerando Tokens para Usuários**

* **Como gerar tokens?**
  + Podemos gerar tokens automaticamente para cada usuário usando um sinal, ou manualmente via comandos no Django.
* **Gerando um Token manualmente**:

python

from rest\_framework.authtoken.models import Token

from django.contrib.auth.models import User

# Gerar um token para um usuário específico

user = User.objects.get(username='username')

token = Token.objects.create(user=user)

print(token.key) # Exibe o token gerado

* **Imagem sugerida**: Captura de tela do Django admin mostrando os tokens gerados para usuários.

**Slide 6: Configurando a Rota para Obtenção do Token**

* **Criar a rota de obtenção do token**:
  + O DRF fornece uma view pronta chamada obtain\_auth\_token para autenticação de token.
* **Exemplo de configuração de URL**:

python

from rest\_framework.authtoken.views import obtain\_auth\_token

from django.urls import path

urlpatterns = [

path('api/token/', obtain\_auth\_token, name='api\_token\_auth'),

]

* **Explicação**:
  + obtain\_auth\_token: Retorna um token de autenticação quando o usuário envia credenciais corretas.
* **Exemplo de Requisição**:
  + **Método**: POST
  + **URL**: /api/token/
  + **Dados**: { "username": "usuario", "password": "senha" }
  + **Resposta esperada**:

json

{

"token": "c8d6b3e7b4a8d2c0f99e3d5c8f7e6c7d"

}

**Slide 7: Introdução ao JWT (JSON Web Token)**

* **O que é JWT?**
  + JWT é um padrão para autenticação baseado em tokens que incluem informações codificadas no próprio token.
* **Como funciona?**:
  + Um JWT é gerado e assinado com uma chave secreta, depois enviado para o cliente.
  + O cliente armazena o token e o inclui em cada requisição.
* **Estrutura do JWT**:
  + **Header**: Algoritmo de criptografia.
  + **Payload**: Dados do usuário.
  + **Signature**: Assinatura única.
* **Diagrama**: Estrutura de um JWT.

mermaid

flowchart LR

Header --> Payload --> Signature

Header[Header: Algoritmo]

Payload[Payload: Dados do Usuário]

Signature[Signature: Assinatura]

**Slide 8: Configurando JWT no Django REST Framework**

* **Instalando a biblioteca JWT**:
  + Comando: pip install djangorestframework-simplejwt
* **Configurando o JWT no DRF**:

python

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

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

],

}

* **Explicação**:
  + JWTAuthentication: Substitui a autenticação por token com JWT.

**Slide 9: Configurando as URLs para JWT**

* **URLs para Obtenção e Renovação do JWT**:
  + O pacote simplejwt inclui views para obtenção e renovação de tokens.
* **Exemplo de configuração de URLs**:

python

from rest\_framework\_simplejwt.views import (

TokenObtainPairView,

TokenRefreshView,

)

from django.urls import path

urlpatterns = [

path('api/token/', TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

path('api/token/refresh/', TokenRefreshView.as\_view(), name='token\_refresh'),

]

* **Descrição**:
  + TokenObtainPairView: View para obter o token de acesso e refresh.
  + TokenRefreshView: View para renovar o token de acesso usando o token de refresh.

**Slide 10: Testando a Autenticação com JWT**

* **Exemplo de Requisição para Obter Token**:
  + **Método**: POST
  + **URL**: /api/token/
  + **Dados**: { "username": "usuario", "password": "senha" }
  + **Resposta esperada**:

json

{

"access": "jwt\_access\_token",

"refresh": "jwt\_refresh\_token"

}

* **Exemplo de Requisição para Renovar Token**:
  + **Método**: POST
  + **URL**: /api/token/refresh/
  + **Dados**: { "refresh": "jwt\_refresh\_token" }
  + **Resposta esperada**:

json

{

"access": "novo\_jwt\_access\_token"

}

**Slide 11: Configurando Permissões na API**

* **Permissões padrão no DRF**:
  + AllowAny: Permite acesso a todos.
  + IsAuthenticated: Apenas usuários autenticados podem acessar.
  + IsAdminUser: Apenas administradores podem acessar.
* **Exemplo de Permissão por Classe de View**:

python

from rest\_framework.permissions import IsAuthenticated

from rest\_framework.views import APIView

class ConsultaView(APIView):

permission\_classes = [IsAuthenticated]

def get(self, request):

...

* **Explicação**:
  + permission\_classes: Define quem pode acessar a view.

**Slide 12: Definindo Permissões Personalizadas**

* **Como criar permissões personalizadas?**:
  + Criamos uma classe que herda de BasePermission e definimos a lógica de permissão.
* **Exemplo de Permissão Personalizada**: Somente o proprietário de um recurso pode editar.

python

from rest\_framework.permissions import BasePermission

class IsOwner(BasePermission):

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

return obj.owner == request.user

* **Explicação**:
  + has\_object\_permission: Método que verifica se o usuário atual é o proprietário do recurso.

**Slide 13: Aplicando Permissões Personalizadas no ViewSet**

* **Exemplo de Uso da Permissão Personalizada** no ViewSet de Consulta.

python

from .permissions import IsOwner

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

permission\_classes = [IsOwner]

* **Explicação**:
  + permission\_classes = [IsOwner]: Apenas o proprietário pode acessar as operações CRUD da consulta.

**Slide 14: Exercício Prático - Configurando Autenticação e Permissão**

* **Tarefa**:
  1. Configure a autenticação JWT na sua API.
  2. Crie uma rota para obter e renovar tokens.
  3. Implemente uma permissão personalizada para garantir que apenas administradores possam excluir recursos.
* **Dicas**:
  + Teste as operações de obtenção e renovação de tokens usando uma ferramenta como Postman.
  + Verifique se as permissões estão funcionando corretamente.

**Slide 15: Recapitulando a Aula**

* **Resumo**:
  + **Autenticação**: Verifica quem está acessando a API.
  + **Autorização**: Limita o que cada usuário pode acessar.
  + **Token Authentication e JWT**: Diferentes métodos de autenticação que adicionam segurança à API.
  + **Permissões**: Controlam o nível de acesso de cada usuário.
* **Benefícios**: Garantia de segurança e controle sobre os dados expostos.

Esses slides fornecem uma abordagem completa para ensinar autenticação e autorização no Django REST Framework, com exemplos práticos, explicações detalhadas e exercícios que ajudam os alunos a compreender e testar o funcionamento das permissões e tokens.

A Aula 6 será dedicada aos conceitos de **Paginação e Filtragem de Dados** no Django REST Framework (DRF). Nessa aula, os alunos aprenderão a configurar a paginação para melhorar a performance de requisições em grandes conjuntos de dados e a implementar filtragem para refinar a busca por informações específicas.

# Aula 6: Paginação e Filtragem de Dados no Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Paginação e Filtragem de Dados no Django REST Framework
* **Objetivo**: Aprender a configurar paginação para gerenciar grandes volumes de dados e implementar filtragem para facilitar a busca de dados específicos na API.

**Slide 2: Introdução à Paginação e Filtragem**

* **Paginação**:
  + Divide grandes conjuntos de dados em partes menores, permitindo uma visualização e carregamento mais rápido.
  + Importante para melhorar a performance e a experiência do usuário em grandes bancos de dados.
* **Filtragem**:
  + Refina os dados com base em critérios específicos (ex.: filtrar por nome, data, etc.).
  + Útil para atender a consultas específicas dos usuários.

**Slide 3: Configuração de Paginação no DRF**

* **Tipos de Paginação**:
  + **PageNumberPagination**: Paginação baseada em número de páginas.
  + **LimitOffsetPagination**: Paginação baseada em limite e offset (ideal para grandes conjuntos de dados).
  + **CursorPagination**: Paginação avançada com base em cursores (útil para dados em ordem específica).
* **Configuração da Paginação** no arquivo settings.py:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10 # Número de itens por página

}

* **Explicação**:
  + DEFAULT\_PAGINATION\_CLASS: Define o tipo de paginação a ser usado.
  + PAGE\_SIZE: Define o número de itens por página.

**Slide 4: Exemplo de Paginação com PageNumberPagination**

* **Descrição**: Paginação simples com base em número de página.
* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?page=2
  + **Resposta**:

json

{

"count": 50,

"next": "http://localhost:8000/api/pacientes/?page=3",

"previous": "http://localhost:8000/api/pacientes/?page=1",

"results": [

{

"id": 11,

"nome": "Maria Silva",

"data\_nascimento": "1990-05-15",

"telefone": "123456789",

"email": "maria@example.com"

},

...

]

}

* **Explicação**:
  + count: Total de registros.
  + next: Link para a próxima página.
  + previous: Link para a página anterior.
  + results: Lista dos dados na página atual.

**Slide 5: Configurando Paginação com LimitOffsetPagination**

* **LimitOffsetPagination**:
  + Usa parâmetros de limite (limit) e offset (offset) para determinar quantos itens exibir e a partir de onde começar.
* **Configuração** no settings.py:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

'PAGE\_SIZE': 10

}

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?limit=5&offset=10
  + **Resposta**: Retorna 5 itens, começando a partir do décimo registro.

**Slide 6: Paginação Avançada com CursorPagination**

* **CursorPagination**:
  + Ideal para APIs que precisam manter a ordem e a consistência dos dados.
  + Usado para dados que mudam frequentemente (ex.: feed de redes sociais).
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.CursorPagination',

'PAGE\_SIZE': 5

}

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?cursor=<valor\_do\_cursor>
  + **Resposta**: Retorna um conjunto de dados com base no cursor.

**Slide 7: Implementando Filtragem Básica no DRF**

* **Por que usar filtragem?**
  + Facilita a consulta de dados específicos com base em parâmetros passados na URL.
* **Instalando o pacote django-filter**:
  + Comando: pip install django-filter
* **Configuração no settings.py**:

python

INSTALLED\_APPS = [

...

'django\_filters',

]

REST\_FRAMEWORK = {

'DEFAULT\_FILTER\_BACKENDS': ['django\_filters.rest\_framework.DjangoFilterBackend']

}

* **Explicação**:
  + DEFAULT\_FILTER\_BACKENDS: Define o backend de filtragem padrão como DjangoFilterBackend.

**Slide 8: Criando Filtros no Serializer**

* **Filtro Básico**:
  + Filtrar pacientes por nome e data\_nascimento.
* **Exemplo de ViewSet**:

python

from django\_filters.rest\_framework import DjangoFilterBackend

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?nome=Maria&data\_nascimento=1990-05-15
  + **Resposta**: Lista apenas os pacientes que correspondem aos parâmetros de nome e data de nascimento.

**Slide 9: Filtros Avançados com filterset\_class**

* **Customizando filtros com filterset\_class**:
  + Permite criar filtros personalizados, com lógica mais complexa.
* **Exemplo de Código para PacienteFilter**:

python

import django\_filters

from .models import Paciente

class PacienteFilter(django\_filters.FilterSet):

nome = django\_filters.CharFilter(lookup\_expr='icontains') # Busca por nome contendo o termo

class Meta:

model = Paciente

fields = ['nome', 'data\_nascimento']

* **Configuração no ViewSet**:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_class = PacienteFilter

* **Explicação**:
  + icontains: Permite realizar uma busca "contendo" para facilitar a filtragem.

**Slide 10: Usando Filtragem com Pesquisa e Ordenação**

* **Habilitando Ordenação**:
  + Podemos adicionar OrderingFilter para ordenar os resultados.
* **Exemplo de Configuração de Ordenação no ViewSet**:

python

from rest\_framework.filters import OrderingFilter

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend, OrderingFilter]

filterset\_class = PacienteFilter

ordering\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição com Ordenação**:
  + URL: http://localhost:8000/api/pacientes/?ordering=nome
  + **Resposta**: Lista de pacientes ordenados pelo nome.

**Slide 11: Exercício Prático - Implementando Paginação e Filtragem**

* **Tarefa**:
  1. Configure LimitOffsetPagination para o modelo Consulta.
  2. Implemente filtros para buscar consultas por data e nome do paciente.
  3. Adicione a possibilidade de ordenar as consultas por data.
* **Passos Sugeridos**:
  + Defina LimitOffsetPagination em settings.py.
  + Configure os filtros e a ordenação no ConsultaViewSet.
  + Teste os filtros e a paginação no navegador ou em uma ferramenta como Postman.

**Slide 12: Benefícios da Paginação e Filtragem**

* **Paginação**:
  + Melhora a performance da API, reduzindo o volume de dados enviados por requisição.
  + Permite aos usuários navegar por conjuntos de dados de forma estruturada.
* **Filtragem**:
  + Facilita a busca de informações específicas sem a necessidade de retornar todos os dados.
  + Oferece flexibilidade na consulta, permitindo resultados mais relevantes.

**Slide 13: Recapitulando a Aula**

* **Resumo**:
  + **Tipos de Paginação**: PageNumberPagination, LimitOffsetPagination, CursorPagination.
  + **Configuração de Filtragem**: Filtros básicos e avançados com DjangoFilterBackend.
  + **Ordenação**: Utilização do OrderingFilter para ordenar resultados.
* **Próximos Passos**: Implementar paginação e filtragem em um projeto prático e testar o funcionamento em endpoints com grande volume de dados.

Com esses slides, você terá uma abordagem completa para ensinar paginação e filtragem no Django REST Framework, com exemplos práticos, explicações detalhadas e exercícios para reforçar o aprendizado.

A Aula 6 será dedicada a ensinar os alunos sobre **Paginação e Filtragem de Dados** no Django REST Framework (DRF). O objetivo é mostrar como lidar com grandes conjuntos de dados, dividindo-os em páginas para melhorar a performance e experiência do usuário e implementando filtragem para refinar as buscas. A aula incluirá diferentes tipos de paginação e técnicas de filtragem detalhadas, com exemplos de código, diagramas e exercícios práticos.

# Aula 6 - repetido (analisar): Paginação e Filtragem de Dados no Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Paginação e Filtragem de Dados no Django REST Framework
* **Objetivo**: Configurar paginação para gerenciar grandes volumes de dados e implementar filtros para facilitar a busca de dados específicos na API, usando ferramentas oferecidas pelo DRF.

**Slide 2: Conceito de Paginação**

* **O que é Paginação?**
  + Técnica que divide um grande conjunto de dados em páginas menores.
  + Melhora a performance ao limitar o número de itens retornados por requisição.
  + Oferece uma experiência de usuário melhor ao permitir navegação por páginas.
* **Por que Usar Paginação?**
  + Reduz o tempo de resposta e o consumo de memória em grandes bancos de dados.
  + Facilita a navegação em interfaces com muitos registros.

**Slide 3: Tipos de Paginação no DRF**

* **Principais Tipos de Paginação**:
  + **PageNumberPagination**: Paginação baseada em número de página (ex.: /pacientes/?page=2).
  + **LimitOffsetPagination**: Define um limite de itens por página e um offset (ex.: /pacientes/?limit=5&offset=10).
  + **CursorPagination**: Utiliza cursores para paginar dados com consistência, útil para dados em ordem.
* **Diagrama de Comparação de Paginação**:

mermaid

flowchart TD

A[PageNumberPagination] --> B[Limite por Página]

A --> C[Específica por Página]

D[LimitOffsetPagination] --> E[Define Limite e Offset]

F[CursorPagination] --> G[Usa Cursores]

G --> H[Ideal para Dados Ordenados]

**Slide 4: Configuração Básica de Paginação no DRF**

* **Definindo Paginação Global no settings.py**:
  + **Configuração** para usar PageNumberPagination com tamanho de página definido.

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10, # Exibe 10 itens por página

}

* **Explicação**:
  + DEFAULT\_PAGINATION\_CLASS: Define o tipo de paginação.
  + PAGE\_SIZE: Define o número de itens por página.

**Slide 5: PageNumberPagination - Exemplo de Uso**

* **PageNumberPagination**:
  + Baseada em número de página.
  + Comum em aplicações com navegação paginada.
* **Exemplo de Requisição**:
  + **URL**: http://localhost:8000/api/pacientes/?page=2
  + **Resposta esperada**:

json

{

"count": 50,

"next": "http://localhost:8000/api/pacientes/?page=3",

"previous": "http://localhost:8000/api/pacientes/?page=1",

"results": [

{

"id": 11,

"nome": "Maria Silva",

"data\_nascimento": "1990-05-15",

"telefone": "123456789",

"email": "maria@example.com"

},

...

]

}

**Slide 6: LimitOffsetPagination - Exemplo de Uso**

* **LimitOffsetPagination**:
  + Define um limite (limit) e um offset (offset) para a página.
  + Útil para APIs onde o cliente controla a quantidade e o ponto de partida.
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

'PAGE\_SIZE': 10

}

* **Exemplo de Requisição**:
  + **URL**: http://localhost:8000/api/pacientes/?limit=5&offset=10
  + **Resposta**: Retorna 5 itens, começando a partir do décimo registro.

**Slide 7: CursorPagination - Exemplo de Uso**

* **CursorPagination**:
  + Baseado em cursores, utiliza um campo específico para ordenação.
  + Ideal para dados que precisam de consistência em ordem específica (ex.: data de criação).
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.CursorPagination',

'PAGE\_SIZE': 5

}

* **Exemplo de Requisição**:
  + **URL**: http://localhost:8000/api/pacientes/?cursor=<valor\_do\_cursor>
  + **Resposta**: Retorna um conjunto de dados baseado no cursor fornecido.

**Slide 8: Customizando a Paginação no DRF**

* **Paginação por Classe Personalizada**:
  + Podemos criar uma classe de paginação para definir comportamentos personalizados.
* **Exemplo de Classe Personalizada**:

python

from rest\_framework.pagination import PageNumberPagination

class PacientePagination(PageNumberPagination):

page\_size = 5 # Número de itens por página

page\_query\_param = 'p' # Parâmetro de página personalizado

max\_page\_size = 50 # Limite máximo de itens

* **Uso no ViewSet**:

python

from .pagination import PacientePagination

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

pagination\_class = PacientePagination

* **Explicação**:
  + page\_size: Define o número de itens por página.
  + page\_query\_param: Define o nome do parâmetro de página na URL.

**Slide 9: Introdução à Filtragem de Dados no DRF**

* **O que é Filtragem?**
  + Refina os dados com base em parâmetros específicos (ex.: filtrar por nome ou data).
  + Facilita a busca e organização dos dados para atender critérios específicos.
* **Exemplo**: Filtrar pacientes por nome ou data\_nascimento.

**Slide 10: Configuração de Filtros com django-filter**

* **Instalando o Pacote**:
  + Comando: pip install django-filter
* **Configuração no settings.py**:

python

INSTALLED\_APPS = [

...

'django\_filters',

]

REST\_FRAMEWORK = {

'DEFAULT\_FILTER\_BACKENDS': ['django\_filters.rest\_framework.DjangoFilterBackend']

}

* **Explicação**:
  + django\_filters: Ferramenta poderosa para definir filtros na API.

**Slide 11: Filtros Básicos no ViewSet**

* **Configurando Filtros Simples no ViewSet**:
  + Filtrar pacientes por nome e data\_nascimento.
* **Exemplo de Código**:

python

from django\_filters.rest\_framework import DjangoFilterBackend

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição**:
  + **URL**: http://localhost:8000/api/pacientes/?nome=Maria&data\_nascimento=1990-05-15
  + **Resultado**: Lista apenas pacientes que atendem aos critérios de filtragem.

**Slide 12: Filtros Avançados com filterset\_class**

* **Por que Usar filterset\_class?**
  + Permite maior controle e complexidade na lógica de filtragem.
* **Exemplo de Código para PacienteFilter**:

python

import django\_filters

from .models import Paciente

class PacienteFilter(django\_filters.FilterSet):

nome = django\_filters.CharFilter(lookup\_expr='icontains') # Busca pelo nome contendo o termo

class Meta:

model = Paciente

fields = ['nome', 'data\_nascimento']

* **Configuração no ViewSet**:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_class = PacienteFilter

**Slide 13: Ordenação de Resultados com OrderingFilter**

* **Ordenação de Resultados**:
  + Podemos adicionar OrderingFilter para permitir que os usuários ordenem os resultados.
* **Configuração de Ordenação**:

python

from rest\_framework.filters import OrderingFilter

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend, OrderingFilter]

filterset\_class = PacienteFilter

ordering\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição**:
  + **URL**: http://localhost:8000/api/pacientes/?ordering=nome
  + **Resposta**: Lista de pacientes ordenados por nome.

**Slide 14: Exercício Prático - Configurando Paginação e Filtragem**

* **Tarefa**:
  1. Configure LimitOffsetPagination para o modelo Consulta.
  2. Implemente filtros para buscar consultas por data e nome do paciente.
  3. Adicione a possibilidade de ordenar as consultas por data.
* **Dicas**:
  + Teste cada funcionalidade usando uma ferramenta como Postman.
  + Combine paginação e filtros para observar a flexibilidade dos recursos.

**Slide 15: Benefícios da Paginação e Filtragem**

* **Paginação**:
  + Reduz o volume de dados retornados em cada requisição.
  + Melhora a performance e a experiência do usuário.
* **Filtragem**:
  + Oferece resultados mais específicos para atender às necessidades dos usuários.
  + Ajuda a tornar a API mais eficiente e flexível.

**Slide 16: Recapitulando a Aula**

* **Resumo**:
  + **Paginação**: Tipos de paginação e configuração no DRF.
  + **Filtragem**: Uso do django-filter para implementar filtros básicos e avançados.
  + **Ordenação**: Como configurar a ordenação de resultados para melhorar a experiência do usuário.
* **Próximos Passos**: Praticar implementando paginação e filtragem em um projeto real.

Com esses slides, você terá uma apresentação completa e detalhada para ensinar Paginação e Filtragem no Django REST Framework, cobrindo configurações, exemplos práticos, diagramas e exercícios que ajudarão os alunos a compreender e aplicar esses recursos no desenvolvimento de APIs.

A Aula 6 focará na **Paginação e Filtragem de Dados** no Django REST Framework (DRF), cobrindo a implementação detalhada de paginação para melhorar a performance de grandes conjuntos de dados e a aplicação de filtros para buscas específicas. Esta aula incluirá instruções práticas, exemplos de código e exercícios.

# Aula 6 - rep: Paginação e Filtragem de Dados no Django REST Framework

**Slide 1: Título e Objetivo da Aula**

* **Título**: Paginação e Filtragem de Dados no Django REST Framework
* **Objetivo**: Implementar paginação para grandes volumes de dados e configurar filtros para buscas específicas, utilizando o DRF para melhorar a eficiência e a usabilidade da API.

**Slide 2: Introdução à Paginação e Filtragem**

* **Paginação**:
  + Técnica que divide grandes conjuntos de dados em partes menores.
  + Melhora a performance e a experiência do usuário.
* **Filtragem**:
  + Refina dados com base em critérios específicos.
  + Facilita a busca de dados específicos sem carregar todos os registros.

**Slide 3: Tipos de Paginação no DRF**

* **PageNumberPagination**: Paginação baseada em números de página.
* **LimitOffsetPagination**: Define um limite de itens e um ponto inicial (offset).
* **CursorPagination**: Usa cursores para navegação ordenada.
* **Diagrama de Comparação**:

mermaid

graph TD

A[PageNumberPagination] --> B[Número de Página]

D[LimitOffsetPagination] --> E[Limite e Offset]

F[CursorPagination] --> G[Cursor para Ordenação]

**Slide 4: Configuração Básica de Paginação no DRF**

* **Definindo Paginação Global no settings.py**:
  + Configuração para usar PageNumberPagination com limite de itens por página.

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

'PAGE\_SIZE': 10

}

* **Explicação**:
  + DEFAULT\_PAGINATION\_CLASS: Define o tipo de paginação.
  + PAGE\_SIZE: Limita o número de itens por página.

**Slide 5: PageNumberPagination - Exemplo Prático**

* **PageNumberPagination**:
  + Baseada em número de página (ex.: ?page=2).
* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?page=2
  + **Resposta esperada**:

json

{

"count": 50,

"next": "http://localhost:8000/api/pacientes/?page=3",

"previous": "http://localhost:8000/api/pacientes/?page=1",

"results": [

{

"id": 11,

"nome": "Maria Silva",

"data\_nascimento": "1990-05-15",

"telefone": "123456789",

"email": "maria@example.com"

},

...

]

}

**Slide 6: LimitOffsetPagination - Exemplo Prático**

* **LimitOffsetPagination**:
  + Usa parâmetros limit e offset para determinar quantos itens exibir e de onde começar.
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.LimitOffsetPagination',

'PAGE\_SIZE': 10

}

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?limit=5&offset=10
  + **Resposta**: Retorna 5 itens a partir do décimo registro.

**Slide 7: CursorPagination - Exemplo Prático**

* **CursorPagination**:
  + Usa cursores para navegação baseada em ordenação.
  + Ideal para dados que precisam de uma ordem consistente (ex.: feeds).
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.CursorPagination',

'PAGE\_SIZE': 5

}

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?cursor=<cursor\_valor>
  + **Resposta**: Retorna dados baseados no cursor.

**Slide 8: Configuração Personalizada de Paginação**

* **Paginação por Classe Personalizada**:
  + Define comportamentos personalizados para paginação.
* **Exemplo de Classe Personalizada**:

python

from rest\_framework.pagination import PageNumberPagination

class PacientePagination(PageNumberPagination):

page\_size = 5

page\_query\_param = 'p'

max\_page\_size = 50

* **Uso no ViewSet**:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

pagination\_class = PacientePagination

**Slide 9: Introdução à Filtragem no DRF**

* **O que é Filtragem?**
  + Refina dados com base em parâmetros específicos.
  + Exemplo: Filtrar pacientes por nome ou data\_nascimento.

**Slide 10: Configurando django-filter para Filtragem**

* **Instalação**:
  + Comando: pip install django-filter
* **Configuração no settings.py**:

python

INSTALLED\_APPS = [

...

'django\_filters',

]

REST\_FRAMEWORK = {

'DEFAULT\_FILTER\_BACKENDS': ['django\_filters.rest\_framework.DjangoFilterBackend']

}

**Slide 11: Implementando Filtros Básicos no ViewSet**

* **Filtro para nome e data\_nascimento**.

python

from django\_filters.rest\_framework import DjangoFilterBackend

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?nome=Maria&data\_nascimento=1990-05-15

**Slide 12: Filtros Avançados com filterset\_class**

* **Por que Usar filterset\_class?**
  + Permite maior controle e lógica personalizada.

python

import django\_filters

from .models import Paciente

class PacienteFilter(django\_filters.FilterSet):

nome = django\_filters.CharFilter(lookup\_expr='icontains') # Busca parcial

class Meta:

model = Paciente

fields = ['nome', 'data\_nascimento']

* **Configuração no ViewSet**:

python

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend]

filterset\_class = PacienteFilter

**Slide 13: Ordenação com OrderingFilter**

* **Ordenação de Resultados**:
  + Adiciona OrderingFilter para ordenar os dados.

python

from rest\_framework.filters import OrderingFilter

class PacienteViewSet(viewsets.ModelViewSet):

queryset = Paciente.objects.all()

serializer\_class = PacienteSerializer

filter\_backends = [DjangoFilterBackend, OrderingFilter]

filterset\_class = PacienteFilter

ordering\_fields = ['nome', 'data\_nascimento']

* **Exemplo de Requisição**:
  + URL: http://localhost:8000/api/pacientes/?ordering=nome

**Slide 14: Exercício Prático - Paginação e Filtragem**

* **Tarefa**:
  1. Configure LimitOffsetPagination no modelo Consulta.
  2. Implemente filtros para buscar consultas por data e paciente.
  3. Adicione ordenação de consultas por data.

**Slide 15: Benefícios da Paginação e Filtragem**

* **Paginação**:
  + Reduz volume de dados por requisição.
* **Filtragem**:
  + Oferece resultados relevantes ao usuário.

**Slide 16: Resumo**

* **Paginação**: Várias opções, incluindo personalizadas.
* **Filtragem**: Configuração básica e avançada.
* **Ordenação**: Controle de ordem dos resultados.

**Aula 6: Finalização**

**Slide 17: Revisão de Conceitos com Diagrama**

* **Diagrama de Fluxo de Paginação e Filtragem**:

mermaid

flowchart LR

Entrada[Entrada do Usuário] -->|Parâmetros de URL| Backend

Backend -->|Aplica Filtros| Filtragem[DjangoFilterBackend]

Backend -->|Aplica Paginação| Paginacao[DRF Pagination]

Paginacao -->|Retorna Dados Paginados| Cliente

* **Resumo**:
  + Parâmetros de URL permitem controle sobre filtros e paginação.
  + Filtragem refina a busca com critérios específicos.
  + Paginação otimiza o volume de dados transferidos.

**Slide 18: Exercício Final de Paginação e Filtragem**

1. **Tarefa Completa**:
   * Implemente um CRUD completo para um modelo de Consulta:
     + Adicione **paginação por página** no ViewSet de Consulta.
     + Configure **filtros** para paciente, médico, e **data de consulta**.
     + Adicione **ordenação** por data.
2. **Desafios**:
   * Adicionar **páginas customizadas** e experimentação com cursores.
   * Testar diferentes configurações para entender o impacto no desempenho.

# Aula 7: Autenticação e Autorização no Django REST Framework

Nesta aula, vamos aprofundar o conceito de **Autenticação** e **Autorização**, essenciais para garantir a segurança da API, definindo quem pode acessar quais dados e de que forma. Veremos diferentes métodos de autenticação, incluindo tokens e JWT (JSON Web Tokens), e como configurar permissões para limitar o acesso.

**Slide 1: Título e Objetivo da Aula**

* **Título**: Autenticação e Autorização no Django REST Framework
* **Objetivo**: Aprender a implementar autenticação segura e permissões de acesso, utilizando métodos como Token e JWT para garantir a segurança da API.

**Slide 2: Introdução à Autenticação e Autorização**

* **Autenticação vs. Autorização**:
  + **Autenticação**: Verifica a identidade do usuário (ex.: login).
  + **Autorização**: Controla o que o usuário autenticado pode fazer.
* **Por que são importantes?**
  + Protegem dados sensíveis e garantem o acesso controlado.
  + Evitam acesso não autorizado a recursos.

**Slide 3: Métodos de Autenticação no DRF**

* **Principais Métodos**:
  + **Token Authentication**: Autenticação baseada em tokens exclusivos.
  + **JWT (JSON Web Tokens)**: Autenticação segura com tokens assinados.
  + **Session Authentication**: Comumente usada para aplicações web.
* **Diagrama Explicativo de Autenticação**:

mermaid

flowchart LR

Cliente -- Envia Credenciais --> Servidor

Servidor -- Retorna Token --> Cliente

Cliente -- Envia Token --> Servidor

Servidor -- Verifica Token --> Recurso[Recursos Protegidos da API]

**Slide 4: Configuração de Autenticação por Token no DRF**

* **Passo a Passo**:
  + Adicione rest\_framework.authtoken em INSTALLED\_APPS.
  + Configure a autenticação por token no settings.py.

python

INSTALLED\_APPS = [

...

'rest\_framework',

'rest\_framework.authtoken',

]

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

'rest\_framework.authentication.TokenAuthentication',

],

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.IsAuthenticated',

],

}

* **Explicação**:
  + TokenAuthentication: Autentica os usuários por meio de tokens.
  + IsAuthenticated: Permite acesso apenas a usuários autenticados.

**Slide 5: Gerando Tokens para Usuários**

* **Como Gerar Tokens?**
  + Tokens podem ser criados automaticamente para novos usuários ou manualmente.

python

from rest\_framework.authtoken.models import Token

from django.contrib.auth.models import User

# Gerando um token para um usuário específico

user = User.objects.get(username='nome\_usuario')

token = Token.objects.create(user=user)

print(token.key)

**Slide 6: Configurando Rota para Obtenção do Token**

* **Utilizando a View Padrão** obtain\_auth\_token:

python

from rest\_framework.authtoken.views import obtain\_auth\_token

from django.urls import path

urlpatterns = [

path('api/token/', obtain\_auth\_token, name='api\_token\_auth'),

]

* **Exemplo de Requisição**:
  + **Método**: POST
  + **URL**: /api/token/
  + **Dados**: { "username": "usuario", "password": "senha" }

**Slide 7: Introdução ao JWT (JSON Web Token)**

* **JWT**:
  + Padrão seguro de autenticação usando tokens assinados.
* **Diagrama Estrutural do JWT**:

mermaid

flowchart LR

Header[Header: Algoritmo e Tipo de Token] --> Payload[Payload: Dados do Usuário]

Payload --> Signature[Signature: Assinatura Segura]

**Slide 8: Configuração de Autenticação JWT**

* **Instalação**:
  + Comando: pip install djangorestframework-simplejwt
* **Configuração no settings.py**:

python

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

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

],

}

**Slide 9: Configurando URLs para JWT**

* **URLs para Obtenção e Renovação**:

python

from rest\_framework\_simplejwt.views import (

TokenObtainPairView,

TokenRefreshView,

)

from django.urls import path

urlpatterns = [

path('api/token/', TokenObtainPairView.as\_view(), name='token\_obtain\_pair'),

path('api/token/refresh/', TokenRefreshView.as\_view(), name='token\_refresh'),

]

* **Explicação**:
  + TokenObtainPairView: Retorna um token de acesso e um token de refresh.
  + TokenRefreshView: Renova o token de acesso.

**Slide 10: Testando a Autenticação JWT**

* **Requisição de Obtenção de Token**:
  + URL: /api/token/
  + **Dados**: { "username": "usuario", "password": "senha" }
* **Resposta Esperada**:

json

{

"access": "jwt\_access\_token",

"refresh": "jwt\_refresh\_token"

}

**Slide 11: Permissões Básicas no DRF**

* **Permissões Padrão**:
  + AllowAny: Acesso para todos os usuários.
  + IsAuthenticated: Acesso apenas para usuários autenticados.
  + IsAdminUser: Acesso apenas para administradores.

**Slide 12: Permissões Personalizadas**

* **Criando Permissões Específicas**:

python

from rest\_framework.permissions import BasePermission

class IsOwner(BasePermission):

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

return obj.owner == request.user

* **Configuração no ViewSet**:

python

class ConsultaViewSet(viewsets.ModelViewSet):

permission\_classes = [IsOwner]

**Slide 13: Exercício Prático - Autenticação e Permissão**

1. **Tarefa Completa**:
   * Implemente a autenticação JWT para a API de Consulta.
   * Adicione permissão personalizada para restringir a edição de consultas ao seu criador.
2. **Dicas**:
   * Use Postman para testar os tokens e permissões.

**Slide 14: Recapitulando a Aula**

* **Autenticação**: Verifica a identidade.
* **Autorização**: Controla o acesso aos recursos.
* **Permissões**: Define o nível de acesso para cada usuário.

Essa estrutura proporciona uma abordagem prática para ensinar autenticação e autorização no Django REST Framework, incluindo exemplos detalhados, diagramas e exercícios para que os alunos possam testar o controle de acesso na API.

# Aula 8: Relacionamentos e Serializers Aninhados

**Objetivo Prático**

Ensinar como lidar com relacionamentos complexos no Django REST Framework (DRF) utilizando serializers aninhados e configurar tipos de relacionamento como **One-to-One**, **One-to-Many** e **Many-to-Many**. A estrutura de exemplo será baseada em um sistema de consultas médicas, com relacionamentos entre **Pacientes**, **Médicos** e **Consultas**.

**Conteúdo Teórico**

1. **Tipos de Relacionamentos**:
   * **One-to-One**: Cada instância de um modelo se relaciona com uma única instância de outro modelo.
   * **One-to-Many**: Uma instância de um modelo pode se relacionar com várias instâncias de outro modelo.
   * **Many-to-Many**: Várias instâncias de um modelo podem se relacionar com várias instâncias de outro modelo.
2. **Serializers Aninhados**:
   * Permitem incorporar dados relacionados de um modelo dentro de outro modelo.
   * Úteis para representar relações complexas entre dados e facilitar o consumo da API.

**Passo a Passo Prático**

**Passo 1: Configuração dos Modelos Relacionados**

1. **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. **Modelo Médico**:
   * 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. **Modelo Consulta com Relacionamentos**:
   * Em consultas/models.py, configure uma relação **Many-to-One** entre Consulta e Paciente e uma **Many-to-One** entre Consulta e Medico.

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: Criando Serializers para Relacionamentos**

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

python

class PacienteSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

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

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

python

class MedicoSerializer(serializers.ModelSerializer):

class Meta:

model = Medico

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

1. **Serializer Aninhado para Consulta**:
   * No serializer de Consulta, aninhe os serializers de Paciente e Medico:

python

class ConsultaSerializer(serializers.ModelSerializer):

paciente = PacienteSerializer() # Aninhando o serializer de Paciente

medico = MedicoSerializer() # Aninhando o serializer de Médico

class Meta:

model = Consulta

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

**Passo 3: Configurando ViewSets e URLs**

1. **ViewSets**:
   * Crie 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. **Configurando 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 os endpoints para consultas. Verifique que as informações de **paciente** e **médico** estão aninhadas dentro do objeto **consulta**.

**Exercício Final**

1. **Tarefa**:
   * Modifique o serializer ConsultaSerializer para permitir criação de consultas com dados de paciente e medico enviados diretamente, sem necessidade de serialização separada.
   * Dica: Use PrimaryKeyRelatedField com write\_only=True.

**Aula 8: Segurança Avançada e Privacidade**

**Objetivo Prático**

Ensinar como implementar práticas de segurança e privacidade para proteger dados sensíveis em uma API REST com Django, abordando métodos como autenticação, controle de acesso, criptografia e conformidade com a LGPD.

**Conteúdo Teórico**

1. **Autenticação Avançada**:
   * **JWT (JSON Web Tokens)** para autenticação segura.
   * Expiração de tokens e renovação automática.
2. **Controle de Acesso**:
   * **Permissões personalizadas** para limitar ações em níveis diferentes.
   * Exemplo: Apenas o **proprietário** de um recurso pode editá-lo.
3. **Proteção de Dados Sensíveis**:
   * **Criptografia** de dados sensíveis em armazenamento.
   * **LGPD**: Implementação de conformidade com a Lei Geral de Proteção de Dados.

**Passo a Passo Prático**

**Passo 1: Configurando Autenticação JWT**

1. **Instalação do JWT**:
   * Instale a biblioteca JWT:

pip install djangorestframework-simplejwt

1. **Configuração no settings.py**:
   * Em settings.py, configure JWTAuthentication:

python

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': [

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

],

}

1. **URLs para Obtenção e Renovação de Tokens**:
   * Em urls.py:

python

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'),

]

**Passo 2: Controle de Acesso com Permissões Personalizadas**

1. **Permissão de Acesso** apenas para o proprietário:
   * Em consultas/permissions.py:

python

from rest\_framework.permissions import BasePermission

class IsOwner(BasePermission):

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

return obj.paciente == request.user

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

python

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

permission\_classes = [IsOwner]

**Passo 3: Proteção de Dados e Privacidade**

1. **Criptografando Campos Sensíveis**:
   * Para dados como **telefone** e **email** de pacientes, utilize uma biblioteca como **django-encrypted-model-fields** (instalação opcional).

pip install django-encrypted-model-fields

1. **Configuração de Privacidade para LGPD**:
   * Explique como anonimizar ou deletar dados sensíveis ao encerrar uma relação contratual, em conformidade com a LGPD.

**Passo 4: Testando Autenticação e Permissões**

1. **Obtenção de Tokens JWT**:
   * Use o endpoint /api/token/ para obter o token com username e password.
   * Use /api/token/refresh/ para renovar o token.
2. **Testando Acesso a Recursos**:
   * No Postman, envie o token no cabeçalho Authorization: Bearer <token>.
   * Verifique que apenas o proprietário de uma consulta pode editá-la.

**Exercício Final**

1. **Implementação de Logs de Acesso**:
   * Configure um log que armazene cada acesso aos recursos da API com data, hora e usuário autenticado.
   * Use logging para registrar em arquivos.

Esses roteiros fornecem um caminho completo para guiar os alunos nas aulas 7 e 8, permitindo que eles implementem relacionamentos complexos e controles de segurança avançados em suas APIs. A prática e os exemplos ajudarão a consolidar o aprendizado e a aplicação desses conceitos fundamentais para a construção de APIs seguras e eficientes.

Roteiro detalhado para a **Aula 8: Segurança Avançada e Privacidade**. Nesta aula, o foco será implementar práticas de segurança no Django REST Framework (DRF), incluindo proteção contra CSRF, autenticação baseada em grupos e ocultação de dados sensíveis. Vamos também explorar como criar permissões personalizadas e controles de acesso detalhados para fortalecer a segurança e privacidade dos dados na API.

# Aula 9: Segurança Avançada e Privacidade

**Objetivo Prático**

Fortalecer as práticas de segurança e configurar requisitos de privacidade em uma API construída com Django REST Framework. Os alunos aprenderão a proteger dados contra-ataques CSRF, a implementar autenticação baseada em grupos de usuários, a ocultar dados sensíveis e a configurar permissões personalizadas para um controle de acesso mais granular.

**Conteúdo Teórico**

**Slide 1: Introdução à Segurança em APIs**

* **Segurança em APIs**:
  + Protege contra-ataques e acessos não autorizados.
  + Garante que apenas usuários autorizados possam acessar e modificar dados.
* **Principais Focos de Segurança**:
  + **Autenticação e Autorização**.
  + **Proteção contra-ataques CSRF e XSS**.
  + **Privacidade e Proteção de Dados Sensíveis**.

**Slide 2: Proteção contra CSRF (Cross-Site Request Forgery)**

* **O que é CSRF?**
  + Um tipo de ataque que leva o usuário autenticado a executar ações indesejadas em uma aplicação em que está autenticado.
* **Proteção Contra CSRF no Django**:
  + O Django possui proteção embutida contra CSRF para formulários e sessões, mas em APIs, onde geralmente usamos tokens, a proteção CSRF não é necessária.
* **Configuração no DRF**:
  + Para APIs sem autenticação de sessão, **desativamos a verificação CSRF** ao usar autenticação baseada em tokens ou JWT.

**Slide 3: Autenticação Baseada em Grupos**

* **Grupos e Permissões**:
  + **Grupos** no Django permitem agrupar usuários com permissões específicas.
  + Útil para controlar o acesso a recursos da API com base nas responsabilidades do usuário (ex.: médicos, recepcionistas, administradores).
* **Configuração de Grupos no Django Admin**:
  + Crie grupos específicos (ex.: Médico, Recepcionista, Administrador) no Django Admin.
  + Atribua permissões de acesso a cada grupo.
* **Diagrama de Controle por Grupos**:
  + Demonstração de acesso diferenciado entre usuários com permissões específicas.

mermaid

graph TD

A[Admin] -->|Permissão Total| API

B[Médico] -->|Permissão Leitura e Escrita| API

C[Recepcionista] -->|Permissão Leitura| API

**Slide 4: Ocultação de Dados Sensíveis**

* **Por que Ocultar Dados Sensíveis?**
  + Dados como senhas, números de cartão, CPF e outros dados pessoais precisam ser protegidos e, se possível, ocultos para atender regulamentações de privacidade como a LGPD.
* **Criptografia e Proteção**:
  + **Criptografia**: Use criptografia para proteger dados sensíveis no banco de dados.
  + **Serializers Condicionais**: Configure serializers para que dados sensíveis não sejam expostos desnecessariamente.
* **Exemplo**:
  + Crie um serializer que omita dados como o CPF ao exibir informações públicas do paciente.

**Prática: Implementando Segurança Avançada e Privacidade**

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

1. **Criando Grupos e Atribuindo Permissões**:
   * Acesse o Django Admin e crie grupos como Medico, Recepcionista e Administrador.
   * Atribua permissões a cada grupo:
     + **Administrador**: Acesso total.
     + **Medico**: Permissão para criar e visualizar consultas.
     + **Recepcionista**: Apenas permissão para visualizar consultas.
2. **Atribuindo Usuários a Grupos**:
   * No Django Admin, associe cada usuário a um grupo específico.

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

1. **Criando uma Permissão Personalizada**:
   * Em consultas/permissions.py, crie uma permissão que restrinja o acesso com base no grupo do usuário.

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, adicione a permissão ao ViewSet de Consulta.

python

from .permissions import IsMedicoOrAdmin

class ConsultaViewSet(viewsets.ModelViewSet):

queryset = Consulta.objects.all()

serializer\_class = ConsultaSerializer

permission\_classes = [IsMedicoOrAdmin]

* + **Explicação**:
    - Apenas usuários que pertencem ao grupo Medico ou são administradores terão acesso total a este endpoint.

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

1. **Criação de Serializers Condicionais**:
   * Em consultas/serializers.py, crie um serializer para Paciente que oculte dados sensíveis, como o CPF, ao exibir informações de leitura pública.

python

class PacientePublicoSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

exclude = ['cpf'] # Oculta o campo CPF

class PacientePrivadoSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

fields = '\_\_all\_\_' # Exibe todos os campos

1. **Aplicando o Serializer Baseado no Tipo de Usuário**:
   * No ViewSet, use o serializer adequado dependendo do grupo de usuários.

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

* + **Explicação**:
    - Usuários do grupo Medico e administradores veem todos os dados, enquanto outros usuários veem apenas dados não sensíveis.

**Passo 4: Testando Autenticação e Permissões**

1. **Teste de Grupos e Permissões no Postman**:
   * Acesse a API como um usuário comum e verifique as permissões de acesso restrito.
   * Teste como um usuário do grupo Medico e como administrador, confirmando os diferentes níveis de acesso.
2. **Verificação de Ocultação de Dados Sensíveis**:
   * Teste o endpoint PacienteViewSet e confirme que dados como o CPF estão ocultos para usuários sem permissão.
   * Verifique que o CPF aparece apenas para usuários médicos ou administradores.

**Exercício Final de Segurança e Privacidade**

1. **Desafio**:
   * Implemente uma permissão personalizada que permita apenas ao médico responsável pela consulta editá-la.
   * Modifique a configuração de ConsultaViewSet para verificar se o usuário é o responsável pela consulta antes de permitir edições.
2. **Dica**:
   * Use has\_object\_permission em uma permissão personalizada que compare o usuário atual com o campo medico no modelo Consulta.

**Slide de Conclusão: Recapitulando a Aula**

* **Resumo**:
  + **CSRF**: Protegido automaticamente em APIs baseadas em tokens.
  + **Autenticação Baseada em Grupos**: Permite controlar o acesso com base em funções específicas de usuário.
  + **Ocultação de Dados Sensíveis**: Proteger dados sensíveis e exibir apenas informações permitidas ao perfil de cada usuário.

Esses roteiros detalhados ajudam os alunos a configurar segurança avançada em suas APIs REST, protegendo dados e definindo permissões de acesso com base em funções, conforme as necessidades específicas de segurança e privacidade do projeto.

Roteiro detalhado para a **Aula 9: Testes Automatizados e Documentação**. Nesta aula, o objetivo será ensinar como garantir a qualidade da API por meio de testes automatizados, usando pytest e unittest, e como documentar a API utilizando ferramentas como **Swagger** e **DRF-YASG**.

# Aula 10: Testes Automatizados e Documentação

**Objetivo Prático**

Ensinar os alunos a implementar testes automatizados para garantir a qualidade e funcionalidade da API e a gerar documentação automatizada para tornar a API acessível e compreensível para outros desenvolvedores.

**Conteúdo Teórico**

**Slide 1: Importância dos Testes Automatizados e da Documentação**

* **Por que testar?**
  + Assegura que a API funcione conforme o esperado e evita regressões.
  + Facilita a identificação de problemas antes que afetem os usuários finais.
* **Por que documentar?**
  + Fornece uma visão clara da API para outros desenvolvedores.
  + Aumenta a facilidade de uso e manutenção da API.

**Slide 2: Tipos de Testes em APIs**

* **Testes Unitários**:
  + Validam partes específicas da lógica do código, como funções e métodos isolados.
* **Testes de Integração**:
  + Verificam se componentes separados da API funcionam corretamente quando integrados.
* **Testes de Aceitação**:
  + Garantem que a API atende aos requisitos funcionais e de negócios.
* **Diagrama de Hierarquia de Testes**:

mermaid

graph TD

Unit[Unitários] --> Integration[Integração]

Integration --> Acceptance[Aceitação]

**Passo a Passo Prático: Testes Automatizados**

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

1. **Instalação do pytest e do pytest-django**:
   * No terminal:

pip install pytest pytest-django

1. **Configuração do pytest**:
   * No diretório do projeto, crie o arquivo 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. **Exemplo de Teste Unitário para o Modelo 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. **Executando os Testes**:
   * No terminal:

pytest

**Passo 3: Criando Testes de Integração com o DRF e pytest**

1. **Exemplo de Teste de Integração para o Endpoint Consulta**:
   * Em consultas/tests/test\_views.py:

python

import pytest

from rest\_framework.test import APIClient

from .models import Consulta, Paciente, Medico

@pytest.mark.django\_db

def test\_consulta\_api():

client = APIClient()

paciente = Paciente.objects.create(nome="Carlos")

medico = Medico.objects.create(nome="Dr. Marcos", crm="54321")

consulta\_data = {

"paciente": paciente.id,

"medico": medico.id,

"data": "2023-11-02",

"horario": "14:00:00"

}

response = client.post("/api/consultas/", consulta\_data, format="json")

assert response.status\_code == 201

assert response.data["data"] == "2023-11-02"

1. **Execução e Verificação**:
   * Execute novamente o pytest e verifique se o teste passou:

pytest

**Documentação da API com Swagger e DRF-YASG**

**Slide 3: Introdução ao Swagger e DRF-YASG**

* **O que é Swagger?**
  + Ferramenta para documentar APIs, permitindo visualização e teste dos endpoints.
* **DRF-YASG**:
  + Biblioteca para gerar automaticamente a documentação Swagger para APIs desenvolvidas com DRF.

**Slide 4: Benefícios da Documentação Automatizada**

* **Facilidade de Uso**: Os desenvolvedores podem visualizar e testar endpoints diretamente.
* **Manutenção**: A documentação é atualizada automaticamente à medida que a API evolui.

**Passo a Passo Prático: Gerando Documentação com Swagger**

**Passo 1: Instalando o DRF-YASG**

1. **Instalação do Pacote**:
   * No terminal:

pip install drf-yasg

1. **Configuração no settings.py**:
   * Certifique-se de que rest\_framework e drf\_yasg estão listados em INSTALLED\_APPS:

python

INSTALLED\_APPS = [

...

'rest\_framework',

'drf\_yasg',

]

**Passo 2: Configuração das URLs para Swagger**

1. **Configurando as Rotas de Documentação**:
   * Em clinic\_api/urls.py, adicione a configuração para gerar a documentação Swagger:

python

from rest\_framework import permissions

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 para o sistema de consultas médicas",

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

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

),

public=True,

permission\_classes=(permissions.AllowAny,),

)

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. **Explicação das URLs**:
   * /swagger/: Gera a interface interativa do Swagger para visualização e teste da API.
   * /redoc/: Gera a documentação da API com a interface do ReDoc.

**Passo 3: Testando a Documentação**

1. **Acessando o Swagger**:
   * Execute o servidor:

python manage.py runserver

* + Acesse a URL http://localhost:8000/swagger/ no navegador para visualizar a interface do Swagger e testar os endpoints.

1. **Acessando o ReDoc**:
   * Acesse http://localhost:8000/redoc/ para ver a documentação no estilo ReDoc, que também é popular para visualização de APIs.

**Exercício Final: Criando Testes e Documentação**

1. **Implementação de Testes para Outro Endpoint**:
   * Peça aos alunos para criar um teste para o endpoint Paciente, validando que um Paciente pode ser criado e visualizado corretamente.
2. **Documentação Personalizada**:
   * Adicione descrições e exemplos nos serializers para melhorar a documentação no Swagger. Por exemplo, no PacienteSerializer:

python

class PacienteSerializer(serializers.ModelSerializer):

class Meta:

model = Paciente

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

extra\_kwargs = {

'nome': {'example': 'Carlos da Silva'},

'data\_nascimento': {'example': '1990-01-01'},

'telefone': {'example': '+55 11 91234-5678'},

}

**Slide de Conclusão: Recapitulando a Aula**

* **Resumo**:
  + **Testes Automatizados**: Garantem que a API funcione corretamente e facilitam a manutenção.
  + **Documentação com Swagger**: Aumenta a acessibilidade e facilita o uso da API por outros desenvolvedores.
  + **Ferramentas Utilizadas**:
    - pytest para testes de integração e unitários.
    - DRF-YASG para geração de documentação automática.

Esses passos práticos ajudam os alunos a desenvolver APIs robustas e bem documentadas, facilitando o desenvolvimento colaborativo e garantindo a qualidade das funcionalidades da API.