

# Apostila 5: Autorização, Permissões Granulares e RBAC Completo (Revisada)

Duração: 4 horas | Nível: Avançado | Pré-requisitos: Apostilas 1, 2, 3 e 4

## 🎯 Objetivos de Aprendizagem

Ao final desta apostila, você será capaz de:

- Diferenciar **Autenticação** (AuthN) de **Autorização** (AuthZ)
- Implementar **Row Level Security** (isolamento de dados por usuário)
- Criar endpoints de **cadastro público** (Sign Up) com segurança
- Gerenciar **grupos e permissões** do Django
- Implementar **RBAC** (Role-Based Access Control) customizado
- Proteger contra **IDOR** (Insecure Direct Object References)
- Criar **permissões dinâmicas** baseadas em regras de negócio

---

## Capítulo 1: O Abismo da Segurança 🚨

### 1.1. O Problema Crítico da Apostila 4

Na apostila anterior, implementamos autenticação JWT. Agora sabemos **quem** é o usuário (`request.user`). Porém, existe uma **falha arquitetural grave**:

```
python

# core/views.py (ESTADO ATUAL - INSEGURO)
class TarefaListCreateAPIView(generics.ListCreateAPIView):
    queryset = Tarefa.objects.all() # ✗ PERIGO!
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated]

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

### O que está errado?

- **GET /api/tarefas/**: João (autenticado) vê TODAS as tarefas do sistema, incluindo as de Maria, Pedro e do CEO
- **Violação de Privacidade**: Dados sensíveis expostos para usuários não autorizados
- **LGPD/GDPR**: Compliance quebrado

## 1.2. Autenticação ≠ Autorização

Esta é a confusão mais comum em segurança de APIs. Vamos esclarecer:

Aspecto	Autenticação (AuthN)	Autorização (AuthZ)
Pergunta	"Quem é você?"	"O que você pode fazer?"
Credencial	Token JWT, Sessão, API Key	Regras, Grupos, Propriedade
Momento	Antes da View executar	Durante a execução da View
Componente DRF	<code>DEFAULT_AUTHENTICATION_CLASSES</code>	<code>permission_classes</code> , <code>get_queryset()</code>
Erro Típico	<code>401 Unauthorized</code>	<code>403 Forbidden</code>
Exemplo	"Token válido do João"	"João só vê suas tarefas"

### Analogia do Mundo Real:

**AUTENTICAÇÃO:** Mostrar seu RG na portaria do prédio  
↓ (Sistema confirma: "Sim, é você mesmo")

**AUTORIZAÇÃO:** Tentar entrar no apartamento 501  
↓ (Sistema verifica: "Você mora no 302, não pode entrar")

## 1.3. O Pipeline de Segurança do DRF

```
REQUISIÇÃO HTTP  
GET /api/tarefas/5/  
Authorization: Bearer <TOKEN>
```

1. AUTHENTICATION | ← JWT valida o token  
(Quem é você?) | ← Define request.user

2. PERMISSION | ← IsAuthenticated?  
(Pode executar?) | ← IsGerente?

3. QUERYSET FILTER | ← get\_queryset()  
(Quais dados ver?) | ← WHERE user\_id = X

4. OBJECT PERMISSION | ← has\_object\_permission()  
(Este objeto?) | ← Dono do recurso?

[RESPOSTA 200]

## Capítulo 2: Row Level Security (Isolamento de Dados)

### 2.1. O Conceito de Multi-Tenancy Lógico

Em aplicações SaaS, cada usuário (ou empresa) vê apenas seus próprios dados, mesmo que tudo esteja no mesmo banco. Isso é **Row Level Security**.

Sem RLS:

```
sql
```

```
SELECT * FROM tarefas;  
-- Retorna: Tarefa do João, Tarefa da Maria, Tarefa do CEO...
```

## Com RLS:

sql

```
SELECT * FROM tarefas WHERE user_id = 5; -- ID do João  
-- Retorna: Apenas tarefas do João
```

## 2.2. Implementando get\_queryset()

O atributo `queryset` é **estático** (definido em tempo de classe). Para lógica **dinâmica** (baseada em quem fez a requisição), usamos o **método** `get_queryset()`.

Edite `core/views.py`:

python

```
# core/views.py  
from rest_framework import generics  
from rest_framework.permissions import IsAuthenticated  
from .models import Tarefa  
from .serializers import TarefaSerializer
```

```
class TarefaListCreateAPIView(generics.ListCreateAPIView):
```

"""

Lista e cria tarefas COM isolamento de dados.  
Cada usuário vê apenas suas próprias tarefas.

"""

```
serializer_class = TarefaSerializer  
permission_classes = [IsAuthenticated]
```

```
def get_queryset(self):
```

"""

SOBRESCREVE o queryset padrão para filtrar por usuário.

Fluxo:

1. JWT decodifica o token → `request.user` (Objeto User)
2. ORM filtra: `WHERE user_id = request.user.id`
3. Retorna QuerySet filtrado

"""

```
# Usuário validado pelo JWTAuthentication  
user = self.request.user
```

```
# Retorna APENAS tarefas deste usuário  
return Tarefa.objects.filter(user=user)
```

```
def perform_create(self, serializer):
```

"""Garante que tarefas criadas pertençam ao usuário logado.""""  
`serializer.save(user=self.request.user)`

## O que mudou:

### Antes

### Depois

```
queryset = Tarefa.objects.all()
```

```
def get_queryset() dinâmico
```

João vê todas as tarefas

João vê apenas suas tarefas

Violação de privacidade

Isolamento completo

## 2.3. Protegendo Operações Individuais (Detail View)

O problema do **IDOR** (Insecure Direct Object References):

João cria tarefa ID=10 (sua)

Maria cria tarefa ID=15 (dela)

João tenta: GET /api/tarefas/15/

Sem proteção:  Sucesso (BUG DE SEGURANÇA!)

Com proteção:  404 Not Found (Correto)

Edita `core/views.py`:

```
python
```

```
# core/views.py (continuação)
```

```
class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
```

```
    """
```

```
    Detalhes, atualização e exclusão de tarefas.
```

```
    Protegido contra IDOR.
```

```
    """
```

```
    serializer_class = TarefaSerializer
```

```
    permission_classes = [IsAuthenticated]
```

```
def get_queryset(self):
```

```
    """
```

```
    Filtra queryset para APENAS tarefas do usuário logado.
```

```
    Segurança:
```

```
    - GET /api/tarefas/999/ (tarefa de outro usuário) → 404
```

```
    - DELETE /api/tarefas/999/ → 404
```

```
    - PUT /api/tarefas/999/ → 404
```

```
Por que 404 e não 403?
```

```
403 Forbidden revela que o recurso existe.
```

```
404 Not Found oculta a existência do dado.
```

```
    """
```

```
    return Tarefa.objects.filter(user=self.request.user)
```

## 2.4. URLs Atualizadas

Edite `core/urls.py`:

```
python
```

```
# core/urls.py
from django.urls import path
from .views import (
    TarefaListCreateAPIView,
    TarefaRetrieveUpdateDestroyAPIView,
)

app_name = 'core'

urlpatterns = [
    # Coleção: Lista e Cria
    path('tarefas/',
        TarefaListCreateAPIView.as_view(),
        name='tarefa-list-create'),

    # Recurso Individual: Detalhe, Atualiza, Deleta
    path('tarefas/<int:pk>/',
        TarefaRetrieveUpdateDestroyAPIView.as_view(),
        name='tarefa-detail'),
]
```

## 2.5. Teste de Isolamento

### Preparação:

1. Crie 2 usuários: `joao` e `maria`
2. Logue com `joao`, crie 2 tarefas (IDs: 1, 2)
3. Logue com `maria`, crie 2 tarefas (IDs: 3, 4)

### Teste 1: GET Lista (João)

```
http
GET /api/tarefas/
Authorization: Bearer <TOKEN_JOAO>

→ 200 OK
[
    {"id": 1, "titulo": "Tarefa do João 1", ...},
    {"id": 2, "titulo": "Tarefa do João 2", ...}
]
```

✓ **Sucesso:** Apenas tarefas do João

### Teste 2: GET Individual - IDOR (João tentando ver tarefa da Maria)

http

GET /api/tarefas/3/

Authorization: Bearer <TOKEN\_JOAO>

→ 404 Not Found

{"detail": "Não encontrado."}

✓ **Sucesso:** Tarefa 3 (da Maria) não é encontrada no queryset do João

### Teste 3: DELETE - IDOR (João tentando deletar tarefa da Maria)

http

DELETE /api/tarefas/4/

Authorization: Bearer <TOKEN\_JOAO>

→ 404 Not Found

✓ **Sucesso:** João não consegue deletar tarefa da Maria

---

## Capítulo 3: Cadastro de Usuários (Sign Up)

### 3.1. O Desafio da Senha

NUNCA salve senhas em texto puro no banco de dados:

python

# ❌ERRADO - VULNERABILIDADE CRÍTICA

```
user = User.objects.create(  
    username='joao',  
    password='123456' # Texto puro no banco!  
)
```

# ✓CORRETO - Usa hashing (bcrypt/PBKDF2)

```
user = User.objects.create_user(  
    username='joao',  
    password='123456' # Será hasheado automaticamente  
)
```

**Comparação:**

Método	Senha no Banco	Segurança
create()	123456 (texto puro)	<span style="color: red;">✗</span> Catastrófica
create_user()	pbkdf2_sha256\$260000\$... (hash)	<span style="color: green;">✓</span> Segura

### 3.2. Serializer de Cadastro

Crie/Edite `core/serializers.py`:

python

```
# core/serializers.py
from django.contrib.auth.models import User
from rest_framework import serializers

class UserRegistrationSerializer(serializers.ModelSerializer):
    """
    Serializer para cadastro de novos usuários.
    """
```

Funcionalidades:

1. Aceita senha na entrada (write\_only)
2. Oculta senha na saída (nunca retorna em JSON)
3. Aplica hashing automático (create\_user)

"""

```
# write_only=True: Campo aceito no POST, mas NUNCA retornado no GET
password = serializers.CharField(
```

```
    write_only=True,
    required=True,
    style={'input_type': 'password'},
    min_length=8, # Segurança: mínimo 8 caracteres
    help_text="Senha com no mínimo 8 caracteres"
)
```

```
# Campo extra para confirmação (opcional, mas recomendado)
password_confirm = serializers.CharField(
```

```
    write_only=True,
    required=True,
    style={'input_type': 'password'}
)
```

```
class Meta:
```

```
    model = User
    fields = ['id', 'username', 'email', 'password', 'password_confirm']
    read_only_fields = ['id']
    extra_kwargs = {
        'email': {'required': True} # Tornar email obrigatório
    }
```

```
def validate(self, data):
    """
    """
```

Validação de objeto completo.

Verifica se as senhas coincidem.

"""

```
if data['password'] != data['password_confirm']:
    raise serializers.ValidationError({
        'password_confirm': 'As senhas não coincidem.'
})
```

```
return data
```

```
def create(self, validated_data):
```

```
    """
```

```
    Sobrescreve o método create para usar create_user.
```

Fluxo:

1. Remove password\_confirm (não faz parte do Model)
2. Extrai a senha
3. Usa create\_user (aplica hash automaticamente)
4. Retorna o usuário criado

```
"""
```

```
# Remove campo extra
```

```
validated_data.pop('password_confirm')
```

```
# Extrai senha
```

```
password = validated_data.pop('password')
```

```
# Cria usuário com hashing
```

```
user = User.objects.create_user(  
    username=validated_data['username'],  
    email=validated_data['email'],  
    password=password  
)
```

```
return user
```

### 3.3. View de Cadastro (Pública)

Esta View **quebra a regra** de segurança padrão: ela deve ser **acessível sem autenticação** (`AllowAny`).

Edite `core/views.py`:

```
python
```

```
# core/views.py
from rest_framework.permissions import AllowAny
from django.contrib.auth.models import User
from .serializers import UserRegistrationSerializer

class RegisterView(generics.CreateAPIView):
    """
    Endpoint PÚBLICO para cadastro de novos usuários.

    Segurança:
    - AllowAny: Qualquer um pode criar conta
    - Senha é hasheada automaticamente
    - Email e username devem ser únicos (validação do Django)

    Exemplo:
    POST /api/register/
    {
        "username": "novo_usuario",
        "email": "novo@exemplo.com",
        "password": "senha_segura_123",
        "password_confirm": "senha_segura_123"
    }
    """

    queryset = User.objects.all()
    permission_classes = [AllowAny] # ← CRÍTICO: Acesso público
    serializer_class = UserRegistrationSerializer
```

## 3.4. URLs do Cadastro

Edite `core/urls.py`:

```
python
```

```
# core/urls.py
from .views import RegisterView # Adicione o import

urlpatterns = [
    # ... (rotas anteriores)

    # Cadastro público
    path('register/',
        RegisterView.as_view(),
        name='register'),
]
```

## 3.5. Teste de Cadastro

## Teste 1: Cadastro Bem-Sucedido

http

POST /api/register/

Content-Type: application/json

```
{  
    "username": "funcionario",  
    "email": "func@empresa.com",  
    "password": "senha123456",  
    "password_confirm": "senha123456"  
}
```

→ 201 Created

```
{  
    "id": 5,  
    "username": "funcionario",  
    "email": "func@empresa.com"  
}
```

**Observação:** A senha NÃO aparece na resposta (write\_only=True)

## Teste 2: Senhas Não Coincidem

http

POST /api/register/

```
{  
    "username": "teste",  
    "email": "teste@exemplo.com",  
    "password": "senha123",  
    "password_confirm": "senha456"  
}
```

→ 400 Bad Request

```
{  
    "password_confirm": ["As senhas não coincidem."]  
}
```

## Teste 3: Username Duplicado

http

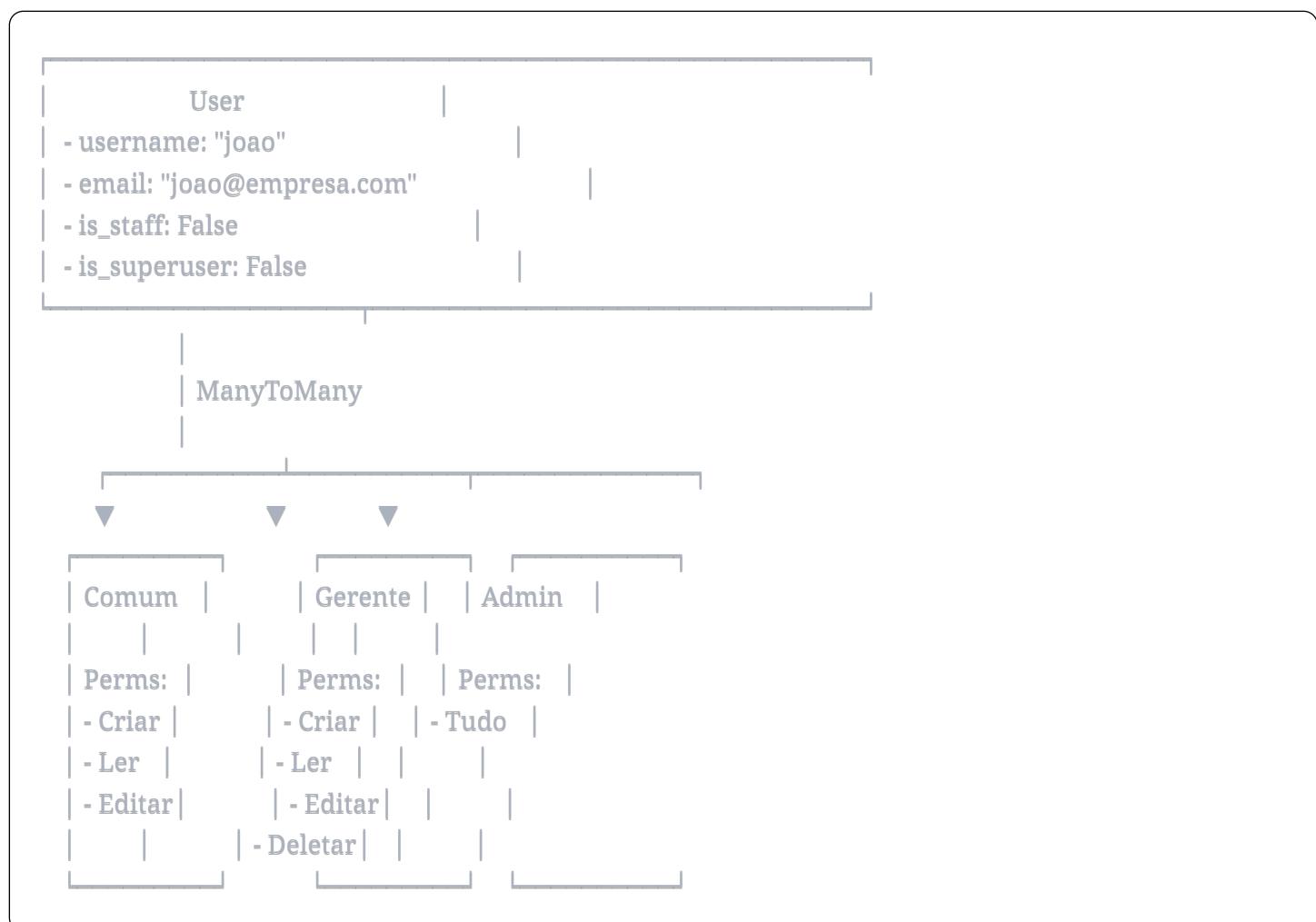
```
POST /api/register/
{
    "username": "funcionario", # Já existe
    "email": "outro@exemplo.com",
    "password": "senha123456",
    "password_confirm": "senha123456"
}

→ 400 Bad Request
{
    "username": ["Usuário com este Nome de usuário já existe."]
}
```

## Capítulo 4: Grupos e Permissões do Django

### 4.1. O Sistema de Grupos do Django

Django vem com um sistema de **grupos** e **permissões** integrado:



### 4.2. Criando Grupos no Sistema

#### Método 1: Via Django Admin

1. Acesse <http://localhost:8000/admin/>
2. Clique em "Grupos"
3. Adicione: **Comum**, **Gerente**, **Admin**

## Método 2: Via Python Shell (Recomendado)

```
bash
```

```
python manage.py shell
```

```
python
```

```
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType
from core.models import Tarefa
```

### # 1. Criar Grupos

```
grupo_comum, _ = Group.objects.get_or_create(name='Comum')
grupo_gerente, _ = Group.objects.get_or_create(name='Gerente')
```

### # 2. Buscar Permissões do Model Tarefa

```
content_type = ContentType.objects.get_for_model(Tarefa)
permissoes = Permission.objects.filter(content_type=content_type)
```

### # 3. Atribuir Permissões ao Grupo Comum

```
perm_add = Permission.objects.get(codename='add_tarefa')
perm_view = Permission.objects.get(codename='view_tarefa')
perm_change = Permission.objects.get(codename='change_tarefa')
```

```
grupo_comum.permissions.add(perm_add, perm_view, perm_change)
```

### # 4. Atribuir TODAS as Permissões ao Gerente

```
perm_delete = Permission.objects.get(codename='delete_tarefa')
grupo_gerente.permissions.add(perm_add, perm_view, perm_change, perm_delete)
```

```
exit()
```

## Método 3: Via Data Migration (Produção)

Crie uma migração:

```
bash
```

```
python manage.py makemigrations --empty core --name create_default_groups
```

Edito o arquivo criado em [core/migrations/000X\\_create\\_default\\_groups.py](#):

```
python
```

```
from django.db import migrations

def create_groups(apps, schema_editor):
    Group = apps.get_model('auth', 'Group')
    Permission = apps.get_model('auth', 'Permission')
    ContentType = apps.get_model('contenttypes', 'ContentType')

    # Buscar ContentType de Tarefa
    try:
        tarefa_ct = ContentType.objects.get(app_label='core', model='tarefa')
    except ContentType.DoesNotExist:
        return

    # Criar Grupos
    grupo_comum, _ = Group.objects.get_or_create(name='Comum')
    grupo_gerente, _ = Group.objects.get_or_create(name='Gerente')

    # Permissões
    perms_comum = Permission.objects.filter(
        content_type=tarefa_ct,
        codename__in=['add_tarefa', 'view_tarefa', 'change_tarefa']
    )

    perms_gerente = Permission.objects.filter(
        content_type=tarefa_ct
    )

    grupo_comum.permissions.set(perms_comum)
    grupo_gerente.permissions.set(perms_gerente)

class Migration(migrations.Migration):
    dependencies = [
        ('core', '0001_initial'), # Ajuste conforme sua migração
    ]

    operations = [
        migrations.RunPython(create_groups),
    ]
```

Aplique:

```
bash
```

```
python manage.py migrate
```

#### 4.3. Atribuindo Grupo Padrão no Cadastro

Vamos modificar o Serializer para que todo usuário novo nasça no grupo "Comum".

Edite `core/serializers.py`:

```
python

# core/serializers.py
from django.contrib.auth.models import User, Group # Adicione Group
from rest_framework import serializers

class UserRegistrationSerializer(serializers.ModelSerializer):
    # ... (campos mantidos iguais)

    def create(self, validated_data):
        """
        Cria usuário E atribui ao grupo padrão "Comum".
        """

        # Remove campo extra
        validated_data.pop('password_confirm')
        password = validated_data.pop('password')

        # Cria usuário
        user = User.objects.create_user(
            username=validated_data['username'],
            email=validated_data['email'],
            password=password
        )

        # 🔍 LÓGICA DE ATRIBUIÇÃO DE CARGO (Default Role)
        try:
            grupo_comum = Group.objects.get(name='Comum')
            user.groups.add(grupo_comum)
        except Group.DoesNotExist:
            # Fallback: Se o grupo não existir, loga erro
            # Em produção, use logging.error() aqui
            pass

        return user
```

## Teste:

1. Crie um usuário via `/api/register/`
2. Vá ao Django Admin
3. Abra o usuário criado
4. Verifique: Na seção "Grupos", deve aparecer "Comum"

# Capítulo 5: RBAC - Role Based Access Control 🎉

## 5.1. Cenário de Negócio

Vamos implementar as seguintes regras:

Cargo	Criar	Ler	Editar	Deletar
Comum	<input checked="" type="checkbox"/> Suas tarefas	<input checked="" type="checkbox"/> Suas tarefas	<input checked="" type="checkbox"/> Suas tarefas	<input type="checkbox"/>
Gerente	<input checked="" type="checkbox"/> Suas tarefas			
Admin	<input checked="" type="checkbox"/> Todas			

## 5.2. Criando Permissão Customizada

Crie o arquivo `core/permissions.py`:

```
python
```

```
# core/permissions.py
from rest_framework import permissions
```

```
class IsGerente(permissions.BasePermission):
```

```
    """
```

Permissão customizada: Acesso concedido apenas a usuários do grupo 'Gerente'.

Uso:

```
permission_classes = [IsAuthenticated, IsGerente]
```

```
"""
```

```
def has_permission(self, request, view):
```

```
    """
```

Verifica se o usuário pertence ao grupo 'Gerente'.

Returns:

bool: True se usuário é Gerente, False caso contrário

```
"""
```

# Validação de segurança: usuário deve estar autenticado

```
if not request.user or not request.user.is_authenticated:
```

```
    return False
```

# Verifica se pertence ao grupo 'Gerente'

```
return request.user.groups.filter(name='Gerente').exists()
```

```
class IsAdminOrOwner(permissions.BasePermission):
```

```
    """
```

Permissão customizada: Acesso concedido a Admins ou ao dono do recurso.

Uso típico: Detail Views (GET, PUT, DELETE de recurso específico)

Regras:

- Staff/Superuser: Acesso total

- Usuário comum: Apenas se for dono (obj.user == request.user)

```
"""
```

```
def has_object_permission(self, request, view, obj):
```

```
    """
```

Verifica permissão em nível de objeto.

Args:

request: Requisição HTTP

view: View sendo executada

obj: Objeto específico (ex: instância de Tarefa)

Returns:

bool: True se usuário é dono, False caso contrário

```
    bool: True se tem permissão, False caso contrário
"""
# Admin/Staff tem acesso total
if request.user.is_staff or request.user.is_superuser:
    return True

# Usuário comum: apenas se for dono
return obj.user == request.user
```

```
class IsOwner(permissions.BasePermission):
```

```
"""
    Permissão simples: Apenas o dono pode acessar.
```

```
Uso: Quando não há exceção para Admin.
"""

```

```
def has_object_permission(self, request, view, obj):
    """Verifica se o usuário é dono do objeto."""
    return obj.user == request.user
```

```
class CanDeleteTask(permissions.BasePermission):
```

```
"""
    Permissão de negócio: Apenas Gerentes podem deletar tarefas.
```

```
Exemplo de regra de negócio customizada.
"""

```

```
message = "Apenas gerentes podem deletar tarefas."
```

```
def has_permission(self, request, view):
    """
        Verifica se a operação é DELETE e se o usuário é Gerente.
    """
    # Se não é DELETE, permite (outras permissões vão validar)
    if request.method != 'DELETE':
        return True

    # Se é DELETE, exige grupo Gerente
    return request.user.groups.filter(name='Gerente').exists()
```

### 5.3. Permissões Dinâmicas por Método HTTP

Vamos modificar a View de detalhe para aplicar permissões diferentes dependendo do método HTTP.

Edite `core/views.py`:

```
python
```

```
# core/views.py
from .permissions import IsGerente, CanDeleteTask # Importe as permissões

class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
    """
```

View de detalhe com RBAC implementado.

Regras:

- GET, PUT, PATCH: Qualquer usuário autenticado (desde que seja dono)
- DELETE: Apenas usuários do grupo 'Gerente'

```
    """
serializer_class = TarefaSerializer
```

```
# Removemos permission_classes estático para usar get_permissions()
```

```
def get_queryset(self):
    """Filtro de isolamento de dados (Row Level Security)."""
    return Tarefa.objects.filter(user=self.request.user)
```

```
def get_permissions(self):
    """
```

Instancia e retorna permissões baseadas no método HTTP.

Fluxo de Verificação:

1. Método é DELETE? → Exige IsAuthenticated + IsGerente
2. Outros métodos? → Apenas IsAuthenticated

Ordem importa:

- IsAuthenticated SEMPRE primeiro (garante login)
- Permissões específicas depois

```
    """
if self.request.method == 'DELETE':
    # DELETE: Precisa estar logado E ser Gerente
    return [IsAuthenticated(), IsGerente()]
```

```
    """
# GET, PUT, PATCH: Apenas logado
# O get_queryset() já garante que só vê suas tarefas
return [IsAuthenticated()]
```

**Alternativa usando CanDeleteTask:**

```
python
```

```
def get_permissions(self):
    """Versão alternativa usando permissão de negócio."""
    # CanDeleteTask já valida se é DELETE + Gerente internamente
    return [IsAuthenticated(), CanDeleteTask()]
```

## 5.4. Protegendo com DjangoModelPermissions

Para usar as permissões nativas do Django (add, view, change, delete):

```
python
```

```
from rest_framework.permissions import DjangoModelPermissions

class TarefaListCreateAPIView(generics.ListCreateAPIView):
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated, DjangoModelPermissions]

    def get_queryset(self):
        return Tarefa.objects.filter(user=self.request.user)

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

### Comportamento:

- POST: Verifica se usuário tem permissão `add_tarefa`
- GET: Verifica se tem permissão `view_tarefa`

---

## Capítulo 6: Testes Completos de Autorização

### 6.1. Preparação do Ambiente

```
bash
```

```
# 1. Criar grupos (se ainda não criou)
python manage.py shell
```

```
python
```

```
from django.contrib.auth.models import Group
Group.objects.get_or_create(name='Comum')
Group.objects.get_or_create(name='Gerente')
exit()
```

```
bash
```

```
# 2. Criar usuários de teste
```

```
python manage.py shell
```

```
python
```

```
from django.contrib.auth.models import User, Group

# Usuário Comum
user_comum = User.objects.create_user('comum', 'comum@test.com', 'senha123')
grupo_comum = Group.objects.get(name='Comum')
user_comum.groups.add(grupo_comum)

# Usuário Gerente
user_gerente = User.objects.create_user('gerente', 'gerente@test.com', 'senha123')
grupo_gerente = Group.objects.get(name='Gerente')
user_gerente.groups.add(grupo_gerente)

exit()
```

## 6.2. Roteiro de Testes

### Teste 1: Cadastro e Atribuição Automática de Grupo

```
http
```

```
POST /api/register/
```

```
Content-Type: application/json
```

```
{
    "username": "novo_usuario",
    "email": "novo@test.com",
    "password": "senha12345678",
    "password_confirm": "senha12345678"
}
```

```
→ 201 Created
```

```
{
    "id": 5,
    "username": "novo_usuario",
    "email": "novo@test.com"
}
```

**Verificação:**

1. Vá ao Django Admin
2. Abra o usuário **novo\_usuario**
3. Verifique: Deve estar no grupo "Comum"

## Teste 2: Isolamento de Dados (Row Level Security)

http

```
# Login como 'comum'  
POST /api/token/  
{"username": "comum", "password": "senha123"}  
→ Guarde o access token
```

```
# Criar tarefa como 'comum'  
POST /api/tarefas/  
Authorization: Bearer <TOKEN_COMUM>  
{"titulo": "Tarefa do Comum", "concluida": false}  
→ 201 Created (ID: 10)
```

```
# Login como 'gerente'  
POST /api/token/  
{"username": "gerente", "password": "senha123"}  
→ Guarde o access token
```

```
# Criar tarefa como 'gerente'  
POST /api/tarefas/  
Authorization: Bearer <TOKEN_GERENTE>  
{"titulo": "Tarefa do Gerente", "concluida": false}  
→ 201 Created (ID: 11)
```

```
# Listar tarefas como 'comum'  
GET /api/tarefas/  
Authorization: Bearer <TOKEN_COMUM>  
→ 200 OK  
[  
 {"id": 10, "titulo": "Tarefa do Comum", ...}  
]
```

 **Sucesso:** Comum vê apenas sua tarefa (ID 10)

## Teste 3: IDOR - Tentativa de Acesso Cruzado

http

```
# 'comum' tenta acessar tarefa do 'gerente' (ID: 11)
```

```
GET /api/tarefas/11/
```

```
Authorization: Bearer <TOKEN_COMUM>
```

→ 404 Not Found

```
{"detail": "Não encontrado."}
```

 **Sucesso:** Bloqueado pelo `get_queryset()`

#### Teste 4: RBAC - Usuário Comum Tenta Deletar

http

```
# 'comum' tenta deletar sua própria tarefa
```

```
DELETE /api/tarefas/10/
```

```
Authorization: Bearer <TOKEN_COMUM>
```

→ 403 Forbidden

```
{
    "detail": "Você não tem permissão para executar essa ação."
}
```

 **Sucesso:** Bloqueado pela permissão `IsGerente`

#### Teste 5: RBAC - Gerente Deleta com Sucesso

http

```
# 'gerente' deleta sua tarefa
```

```
DELETE /api/tarefas/11/
```

```
Authorization: Bearer <TOKEN_GERENTE>
```

→ 204 No Content

 **Sucesso:** Gerente tem permissão

#### Teste 6: Gerente NÃO Pode Deletar Tarefa de Outro

http

```
# Criar nova tarefa como 'comum'  
POST /api/tarefas/  
Authorization: Bearer <TOKEN_COMUM>  
{"titulo": "Tarefa protegida", "concluida": false}  
→ 201 Created (ID: 12)
```

```
# 'gerente' tenta deletar tarefa do 'comum'  
DELETE /api/tarefas/12/  
Authorization: Bearer <TOKEN_GERENTE>  
  
→ 404 Not Found
```

✓ **Sucesso:** `(get_queryset()` impede acesso cruzado

## Capítulo 7: Casos Avançados e Exceções

### 7.1. Admin Acessa Tudo (IsAdminOrOwner)

Para permitir que Staff/Superuser acessem dados de todos os usuários (para suporte técnico):

```
python  
  
# core/views.py  
  
class TarefaListCreateAPIView(generics.ListCreateAPIView):  
    serializer_class = TarefaSerializer  
    permission_classes = [IsAuthenticated]  
  
    def get_queryset(self):  
        """  
        Lógica condicional:  
        - Admin/Staff: Vê tudo  
        - Usuário comum: Vê apenas suas tarefas  
        """  
        user = self.request.user  
  
        # Exceção para Staff  
        if user.is_staff or user.is_superuser:  
            return Tarefa.objects.all()  
  
        # Isolamento para usuários comuns  
        return Tarefa.objects.filter(user=user)  
  
    def perform_create(self, serializer):  
        serializer.save(user=self.request.user)
```

## 7.2. Permissão por Objeto (has\_object\_permission)

Para regras mais complexas que dependem do estado do objeto:

```
python  
  
# core/permissions.py  
  
class CanEditOnlyPendingTasks(permissions.BasePermission):  
    """  
        Regra de Negócio: Tarefas concluídas não podem ser editadas.  
    """  
  
    message = "Tarefas concluídas não podem ser editadas."  
  
    def has_object_permission(self, request, view, obj):  
        # Leitura sempre permitida  
        if request.method in permissions.SAFE_METHODS:  
            return True  
  
        # Edição: apenas se tarefa não estiver concluída  
        return not obj.concluida
```

Uso na View:

```
python  
  
class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):  
    serializer_class = TarefaSerializer  
    permission_classes = [IsAuthenticated, CanEditOnlyPendingTasks]  
  
    def get_queryset(self):  
        return Tarefa.objects.filter(user=self.request.user)
```

## 7.3. Permissões Compostas (AND / OR)

**Operador AND (Todas devem passar):**

```
python  
  
permission_classes = [IsAuthenticated, IsGerente, IsOwner]  
# Precisa: Estar logado E Ser Gerente E Ser dono
```

**Operador OR (Pelo menos uma deve passar):**

```
python
```

```
from rest_framework.permissions import OR

class MyView(APIView):
    def get_permissions(self):
        return [OR(IsGerente(), IsAdminUser())]
    # Precisa: Ser Gerente OU Ser Admin
```

## 7.4. Permissões em Actions (ViewSets)

Se estiver usando ViewSets (veremos em apostilas futuras):

```
python
```

```
from rest_framework import viewsets
from rest_framework.decorators import action

class TarefaViewSet(viewsets.ModelViewSet):
    serializer_class = TarefaSerializer

    def get_permissions(self):
        """Permissões por action."""
        if self.action == 'destroy':
            return [IsAuthenticated(), IsGerente()]
        return [IsAuthenticated()]

    @action(detail=False, methods=['get'], permission_classes=[IsGerente])
    def estatisticas(self, request):
        """Endpoint exclusivo para Gerentes."""
        # ...
```

## Capítulo 8: Endpoint "Meu Perfil" (/api/me/)

### 8.1. Serializer de Perfil

Crie `core/serializers.py`:

```
python
```

```
# core/serializers.py

class UserProfileSerializer(serializers.ModelSerializer):
    """
    Serializer para exibir dados do usuário logado.
    Inclui informações sobre grupos.
    """

    # Campo customizado: lista de nomes dos grupos
    grupos = serializers.SerializerMethodField()

    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'first_name', 'last_name',
                  'is_staff', 'date_joined', 'grupos']
        read_only_fields = ['id', 'username', 'is_staff', 'date_joined']

    def get_grupos(self, obj):
        """
        Retorna lista de nomes dos grupos do usuário.

        Returns:
            list: ['Comum', 'Gerente']
        """

        return [grupo.name for grupo in obj.groups.all()]
```

## 8.2. View de Perfil

```
python
```

```
# core/views.py
from rest_framework.views import APIView
from rest_framework.response import Response
```

```
class MeView(APIView):
```

```
    """
```

```
    Endpoint para visualizar dados do próprio usuário.
```

```
    GET /api/me/ → Retorna dados do usuário logado
```

```
    """
```

```
    permission_classes = [IsAuthenticated]
```

```
    def get(self, request):
```

```
        """
```

```
        Retorna perfil do usuário autenticado.
```

```
        """
```

```
        user = request.user
```

```
        serializer = UserProfileSerializer(user)
```

```
        return Response(serializer.data)
```

```
# Alternativa usando RetrieveAPIView (mais elegante)
```

```
class MeRetrieveView(generics.RetrieveAPIView):
```

```
    """
```

```
    Endpoint para visualizar dados do próprio usuário.
```

```
    Versão usando generics (mais DRF-style).
```

```
    """
```

```
    serializer_class = UserProfileSerializer
```

```
    permission_classes = [IsAuthenticated]
```

```
    def get_object(self):
```

```
        """
```

```
        Sobrescreve para retornar o usuário da requisição.
```

```
        Não usa pk da URL, usa request.user.
```

```
        """
```

```
        return self.request.user
```

## 8.3. URL

```
python
```

```
# core/urls.py
from .views import MeRetrieveView

urlpatterns = [
    # ... (rotas anteriores)

    # Perfil do usuário logado
    path('me/',
        MeRetrieveView.as_view(),
        name='user-profile'),
]

]
```

## 8.4. Teste

```
http
```

```
GET /api/me/
Authorization: Bearer <ACCESS_TOKEN>
```

```
→ 200 OK
{
    "id": 5,
    "username": "funcionario",
    "email": "func@empresa.com",
    "first_name": "",
    "last_name": "",
    "is_staff": false,
    "date_joined": "2025-12-16T10:30:00Z",
    "grupos": ["Comum"]
}
```

## Capítulo 9: Mudança de Senha 🔑

### 9.1. Serializer de Mudança de Senha

python

```
# core/serializers.py

class ChangePasswordSerializer(serializers.Serializer):
    """
    Serializer para mudança de senha.
    Não é um ModelSerializer pois não salva diretamente no Model.
    """

    old_password = serializers.CharField(
        required=True,
        write_only=True,
        style={'input_type': 'password'}
    )
    new_password = serializers.CharField(
        required=True,
        write_only=True,
        min_length=8,
        style={'input_type': 'password'}
    )
    new_password_confirm = serializers.CharField(
        required=True,
        write_only=True,
        style={'input_type': 'password'}
    )

    def validate_old_password(self, value):
        """Valida se a senha antiga está correta."""
        user = self.context['request'].user
        if not user.check_password(value):
            raise serializers.ValidationError("Senha atual incorreta.")
        return value

    def validate(self, data):
        """Valida se as novas senhas coincidem."""
        if data['new_password'] != data['new_password_confirm']:
            raise serializers.ValidationError({
                'new_password_confirm': 'As senhas não coincidem.'
            })
        return data

    def save(self, **kwargs):
        """Atualiza a senha do usuário."""
        user = self.context['request'].user
        user.set_password(self.validated_data['new_password'])
        user.save()
        return user
```

## 9.2. View de Mudança de Senha

```
python
```

```
# core/views.py

class ChangePasswordView(generics.UpdateAPIView):
    """
    Endpoint para mudança de senha do usuário logado.

    POST /api/change-password/
    {
        "old_password": "senha_antiga",
        "new_password": "senha_nova_123",
        "new_password_confirm": "senha_nova_123"
    }
    """

    serializer_class = ChangePasswordSerializer
    permission_classes = [IsAuthenticated]

    def get_object(self):
        """
        Retorna o usuário logado.
        """
        return self.request.user

    def update(self, request, *args, **kwargs):
        """
        Sobrescreve update para customizar resposta.
        """

        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        return Response({
            'detail': 'Senha alterada com sucesso. Por favor, faça login novamente.'
        }, status=status.HTTP_200_OK)
```

## 9.3. URL

```
python
```

```
# core/urls.py
from .views import ChangePasswordView

urlpatterns = [
    # ... (rotas anteriores)

    # Mudança de senha
    path('change-password/',
        ChangePasswordView.as_view(),
        name='change-password'),
]
```

## 9.4. Teste

```
http
```

```
POST /api/change-password/
Authorization: Bearer <ACCESS_TOKEN>
Content-Type: application/json
```

```
{
    "old_password": "senha123456",
    "new_password": "nova_senha_segura_789",
    "new_password_confirm": "nova_senha_segura_789"
}
```

```
→ 200 OK
```

```
{
    "detail": "Senha alterada com sucesso. Por favor, faça login novamente."
}
```

## Teste de Erro:

http

```
POST /api/change-password/  
Authorization: Bearer <ACCESS_TOKEN>  
  
{  
    "old_password": "senha_errada",  
    "new_password": "nova_senha",  
    "new_password_confirm": "nova_senha"  
}  
  
→ 400 Bad Request  
{  
    "old_password": ["Senha atual incorreta."]  
}
```

## Exercícios Práticos

### Exercício 1: Endpoint de Estatísticas

Crie um endpoint `/api/stats/` que retorna estatísticas do usuário:

python

```
# core/views.py  
  
class UserStatsView(APIView):  
    """  
    GET /api/stats/ → Estatísticas do usuário logado  
    """  
    permission_classes = [IsAuthenticated]  
  
    def get(self, request):  
        user = request.user  
        tarefas = Tarefa.objects.filter(user=user)  
  
        total = tarefas.count()  
        concluidas = tarefas.filter(concluida=True).count()  
        pendentes = total - concluidas  
        taxa_conclusao = (concluidas / total * 100) if total > 0 else 0  
  
        return Response({  
            'total_tarefas': total,  
            'concluidas': concluidas,  
            'pendentes': pendentes,  
            'taxa_conclusao': round(taxa_conclusao, 2)  
        })
```

## Teste Esperado:

```
http
```

```
GET /api/stats/
```

```
Authorization: Bearer <TOKEN>
```

```
→ 200 OK
```

```
{  
    "total_tarefas": 10,  
    "concluidas": 6,  
    "pendentes": 4,  
    "taxa_conclusao": 60.0  
}
```

## Exercício 2: Bloqueio de Edição de Email

Modifique `UserProfileSerializer` para impedir mudança de email:

```
python
```

```
class UserProfileSerializer(serializers.ModelSerializer):  
    # ... (campos anteriores)  
  
    class Meta:  
        model = User  
        fields = ['id', 'username', 'email', 'first_name', 'last_name', 'grupos']  
        read_only_fields = ['id', 'username', 'email'] # ← Email read-only
```

## Crie View de Atualização:

```
python
```

```
class UpdateProfileView(generics.UpdateAPIView):  
    serializer_class = UserProfileSerializer  
    permission_classes = [IsAuthenticated]  
  
    def get_object(self):  
        return self.request.user
```

## Exercício 3: Permissão "Apenas Admin Deleta Usuários"

Crie permissão e endpoint:

```
python
```

```
# core/permissions.py
class IsAdminUser(permissions.BasePermission):
    def has_permission(self, request, view):
        return request.user.is_staff

# core/views.py
class UserDestroyView(generics.DestroyAPIView):
    """DELETE /api/users/<pk>/ → Apenas Admin pode deletar"""
    queryset = User.objects.all()
    permission_classes = [IsAuthenticated, IsAdminUser]
```

## Exercício 4: Tarefas Compartilhadas (Avançado)

Adicione campo `compartilhada` ao Model:

```
python
```

```
# core/models.py
class Tarefa(models.Model):
    # ... (campos existentes)
    compartilhada = models.BooleanField(default=False)
```

Modifique `get_queryset()`:

```
python
```

```
from django.db.models import Q

def get_queryset(self):
    user = self.request.user

    # Retorna: Tarefas próprias OU tarefas compartilhadas
    return Tarefa.objects.filter(
        Q(user=user) | Q(compartilhada=True)
    )
```

## 🎓 Resumo da Apostila

O que aprendemos:

**Autenticação vs Autorização:** Diferença entre "quem é" e "o que pode fazer"  **Row Level Security:** Isolamento de dados por usuário (`get_queryset()`)  **Proteção IDOR:** IDs de outros usuários retornam 404 em vez de 403  **Cadastro Seguro:** Hashing de senhas com `create_user()`  **Grupos Django:** Sistema nativo de cargos e permissões  **Atribuição Automática:** Novos usuários nascem com cargo padrão  **RBAC Customizado:** Permissões baseadas em regras de negócio  **Permissões Dinâmicas:** `get_permissions()` por método HTTP  **Endpoints Auxiliares:** `/me/`, `/change-password/`, `/stats/`

## Fluxo de Segurança Completo:

1. JWT valida token → `request.user` definido
2. `IsAuthenticated` verifica login
3. `IsGerente/outras` permissões verificam cargo
4. `get_queryset()` filtra dados por usuário
5. `has_object_permission()` valida objeto específico
6. View executa lógica de negócio
7. Response retornada

## 🔍 Checklist de Verificação

Antes de prosseguir para a Apostila 6:

- `get_queryset()` filtra tarefas por usuário logado
- Tentativa de acesso a ID de outro usuário retorna 404
- Endpoint `/api/register/` funciona sem autenticação
- Usuários novos nascem no grupo "Comum" automaticamente
- Grupos "Comum" e "Gerente" existem no banco
- Usuários comuns NÃO conseguem deletar tarefas
- Gerentes conseguem deletar suas tarefas
- Gerentes NÃO conseguem deletar tarefas de outros
- Endpoint `/api/me/` retorna dados do usuário logado
- Mudança de senha valida senha antiga corretamente
- Entende diferença entre `has_permission` e `has_object_permission`



Sua API agora possui **segurança robusta em múltiplas camadas**:

1. **Autenticação:** JWT valida identidade
2. **Autorização:** Permissões controlam ações
3. **Isolamento:** Cada usuário vê apenas seus dados
4. **RBAC:** Cargos definem poderes diferentes
5. **Proteção IDOR:** IDs externos retornam 404

Na **Apostila 6**, vamos garantir que tudo isso continue funcionando no futuro através de **Testes Automatizados** (Unitários e de Integração). Vamos escrever testes para cada regra de segurança implementada!