Apostila 5: Autorização Granular, Cadastro de Usuários e RBAC (Role-Based Access Control)

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

Objetivos de Aprendizagem

Ao final desta apostila, você dominará:

- **Diferença Crítica entre AuthN e AuthZ:** Entender profundamente a separação entre saber quem é o usuário e o que ele pode fazer.
- **Row Level Security (RLS):** Implementar filtros no QuerySet para garantir que dados sejam isolados por usuário (Multi-tenancy lógico).
- **Cadastro Público (Sign Up):** Criar endpoints abertos (AllowAny) para registro de novos usuários com segurança de senha (hashing).
- **Gestão de Senhas:** Uso correto de write_only e create_user para proteção criptográfica.
- **Atribuição Automática de Cargos:** Uso de *Signals* ou lógica no Serializer para definir grupos padrão (Default Role).
- **Permissões Customizadas (RBAC):** Criar classes de permissão do zero para regras de negócio complexas (ex: "Apenas Gerentes deletam").

Capítulo 1: O Abismo da Segurança (Autenticação $\neq$ Autorização)

1.1. O Cenário Atual (Apostila 4)

Na apostila anterior, configuramos o JWT. O sistema agora sabe *quem* é o usuário através do token (request.user). No entanto, nosso código em views.py ainda possui uma falha arquitetural grave para aplicações SaaS:

Python

```
# core/views.py (Estado Atual)
class TarefaListCreateAPIView(generics.ListCreateAPIView):
    queryset = Tarefa.objects.all()  # <--- O PERIGO MORA AQUI
```

```
    # ...
```

Ao usar .all(), dizemos ao Django: "Traga **todos** os registros da tabela core_tarefa".

- **Cenário de Ataque:** Se o usuário "João" (autenticado) fizer um GET em /api/tarefas/, ele receberá um JSON contendo as tarefas da "Maria", do "Pedro" e do "CEO".
- **Consequência:** Violação de privacidade e LGPD.

1.2. A Matriz de Diferenças: AuthN vs AuthZ

É vital não confundir os dois conceitos. O DRF trata eles em etapas distintas do pipeline de requisição.

Característica	Autenticação (AuthN)	Autorização (AuthZ)
Pergunta Fundamental	"Quem é você?"	"Você tem permissão para tocar nisso?"
Credencial	Token (JWT), Sessão, Basic Auth	Regras de Negócio, Grupos, Propriedade
Momento no DRF	Executa **antes** da View	Executa **dentro** da View (ou antes do método)
Componente DRF	DEFAULT_AUTHENTICATION_CLASSES	permission_classes e get_queryset
Erro Padrão	401 Unauthorized (Não sei quem é você)	403 Forbidden (Sei quem é, mas saia daqui)

Capítulo 2: Row Level Security (Meus Dados, Minhas Regras)

Para resolver o problema do Capítulo 1, precisamos implementar o isolamento de dados. Em vez de bloquear o acesso na View, nós **filtramos** os dados no banco.

2.1. Sobrescrevendo get_queryset

No DRF, o atributo queryset é estático. Para lógica dinâmica (baseada no request.user), devemos usar o método get_queryset().

Mão na Massa: Edite core/views.py:

Python

```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):
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated]  # Exige Token válido

    def get_queryset(self):
        """
        Sobrescreve o comportamento padrão para retornar APENAS
        os dados pertencentes ao usuário logado.
        """
        # 1. Recupera o usuário validado pelo JWT
        user = self.request.user

        # 2. Retorna o filtro. O Django fará o WHERE user_id = X no banco.
        return Tarefa.objects.filter(user=user)

    def perform_create(self, serializer):
        # Garante que a tarefa criada seja vinculada ao usuário logado
        serializer.save(user=self.request.user)
```

2.2. O Buraco do ID (Insecure Direct Object References - IDOR)

Ainda temos um problema. Se o João tentar acessar /api/tarefas/50/ (onde o ID 50 é da Maria), e a view de detalhe usar queryset = Tarefa.objects.all(), ele conseguirá acessar.

Precisamos aplicar o mesmo filtro na View de Detalhe.

Python

```python
# core/views.py (continuação)

class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        """
        Garante que operações de detalhe (GET, PUT, DELETE por ID)
        só encontrem o objeto se ele pertencer ao usuário.
        """
        user = self.request.user
        return Tarefa.objects.filter(user=user)
```

2.3. Análise de Segurança: 404 vs 403

Com a implementação acima (get_queryset filtrado), o comportamento do sistema muda de forma interessante:

- **Tentativa:** João acessa /api/tarefas/50/ (Tarefa da Maria).
- **Comportamento:** O Django tenta buscar o ID 50 **dentro** da lista filtrada do João.
- **Resultado:** O ID 50 não existe na lista do João.
- **Resposta:** 404 Not Found.

Por que isso é bom?
Se retornássemos 403 Forbidden, estaríamos confirmando para o atacante que "O ID 50 existe, mas você não pode ver". Ao retornar 404, ocultamos a própria existência do dado.

Capítulo 3: Cadastro de Usuários (Sign Up)

Até o momento, dependemos do terminal (createsuperuser) para criar contas. Uma API real precisa de um endpoint público para cadastro ("Registrar-se").

3.1. O Desafio da Senha (Hashing)

Não podemos usar um ModelSerializer comum para o model User. Se fizermos isso, a senha será salva em **texto puro** no banco de dados, o que é uma falha crítica de segurança. Precisamos usar o método auxiliar create_user do Django, que aplica o hash (criptografia) automaticamente.

Mão na Massa: Crie/Edite core/serializers.py.

Python

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

class UserRegistrationSerializer(serializers.ModelSerializer):
    # Definimos 'write_only=True' para que a senha seja aceita no cadastro (POST),
    # mas NUNCA seja devolvida na resposta (Response JSON).
    password = serializers.CharField(
        write_only=True,
        required=True,
        style={'input_type': 'password'}
    )

    class Meta:
        model = User
        fields = ['username', 'email', 'password']

    def create(self, validated_data):
        """
        Intercepta a criação para usar o 'create_user' e hashear a senha.
        """
        # Extrai a senha dos dados validados
        password = validated_data.pop('password')

        # Extrai email e username
        email = validated_data.get('email', '')
        username = validated_data['username']

        # Cria a instância usando o método seguro do Django
        user = User.objects.create_user(
            username=username,
            email=email,
            password=password
        )
        return user
```

3.2. A View de Cadastro (Pública)

Esta View deve quebrar a regra padrão de segurança: ela deve ser acessível por qualquer

pessoa (AllowAny), afinal, o usuário ainda não tem conta para se autenticar.

Mão na Massa: Edite core/views.py.

Python

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

class RegisterView(generics.CreateAPIView):
    """
    Endpoint para cadastro de novos usuários.
    Acesso: Público (Qualquer um pode criar conta).
    """
    queryset = User.objects.all()
    permission_classes = [AllowAny] # Sobrescreve o padrão global
    serializer_class = UserRegistrationSerializer
```

3.3. Configurando a Rota

Edite core/urls.py (ou config/urls.py).

Python

```
# core/urls.py
from django.urls import path
from .views import (
    TarefaListCreateAPIView,
    TarefaRetrieveUpdateDestroyAPIView,
    RegisterView # <-- Importe a nova view
)

urlpatterns = [
    path('tarefas/', TarefaListCreateAPIView.as_view(), name='tarefas-list'),
    path('tarefas/<int:pk>/', TarefaRetrieveUpdateDestroyAPIView.as_view(),
name='tarefas-detail'),
```

```
    # Nova rota de registro
    path('register/', RegisterView.as_view(), name='register'),
]
```

Capítulo 4: Cargos e Grupos (Default Roles)

Em sistemas profissionais, raramente um usuário novo tem "acesso total" ou "nenhum acesso". Geralmente, ele é alocado em um Grupo Padrão (ex: "Colaborador" ou "Free Tier").

Vamos automatizar isso para que todo usuário criado via API receba o cargo "Comum".

4.1. Criando os Grupos (Pré-requisito)

Os grupos precisam existir no banco de dados. Você pode criá-los via Admin ou via Python Shell. Vamos garantir que existam.

Abra o terminal:

```
Bash
```

```
python manage.py shell
```

No shell interativo do Django:

```
Python
```

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

4.2. Injetando o Grupo no Cadastro

Voltamos ao UserRegistrationSerializer para modificar o método create.

Mão na Massa: Atualize core/serializers.py.

Python

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

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

    def create(self, validated_data):
        # 1. Cria o usuário com segurança
        password = validated_data.pop('password')
        user = User.objects.create_user(
            username=validated_data['username'],
            email=validated_data.get('email', ''),
            password=password
        )

        # 2. Lógica de Atribuição de Cargo (Role)
        try:
            # Busca o grupo 'Comum'
            grupo_comum = Group.objects.get(name='Comum')
            # Adiciona o usuário ao grupo
            user.groups.add(grupo_comum)
        except Group.DoesNotExist:
            # Fallback: Se o grupo não existir, o usuário é criado sem grupo.
            # Em produção, deveríamos logar um erro aqui.
            pass

        return user
```

Agora, todo usuário nascido via API /api/register/ terá automaticamente as permissões associadas ao grupo "Comum".

Capítulo 5: Permissões Avançadas (RBAC - Role Based

Access Control)

Vamos implementar uma regra de negócio complexa para justificar o uso de cargos:

- **Usuários Comuns:** Podem Ler (GET), Criar (POST) e Editar (PUT/PATCH) suas tarefas.
- **Gerentes:** Têm todos os poderes dos Comuns, mais o poder exclusivo de **DELETAR** tarefas.

5.1. Criando uma Permissão Customizada (IsGerente)

O DRF permite criar classes de permissão puras. Crie um novo arquivo core/permissions.py.

Python

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

class IsGerente(permissions.BasePermission):
    """
    Permissão customizada que concede acesso apenas se o usuário
    pertencer ao grupo 'Gerente'.
    """

    def has_permission(self, request, view):
        # 1. Verificação de Sanidade: Usuário deve estar logado
        if not request.user or not request.user.is_authenticated:
            return False

        # 2. Verificação de Grupo: Checa se 'Gerente' está na lista de grupos
        return request.user.groups.filter(name='Gerente').exists()
```

5.2. View Dinâmica (Permissões por Verbo HTTP)

Agora precisamos dizer à View: "Se for DELETE, exija IsGerente. Se for outro método, apenas IsAuthenticated".

Mão na Massa: Edite core/views.py.

Python

```python
# core/views.py
from .permissions import IsGerente # Importe sua permissão customizada

class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = TarefaSerializer
    # Removemos a linha estática 'permission_classes' para usar o método dinâmico

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

    def get_permissions(self):
        """
        Instancia e retorna a lista de permissões que esta view requer,
        dependendo do método HTTP da requisição.
        """
        if self.request.method == 'DELETE':
            # Para deletar: Precisa estar logado E ser Gerente
            # A ordem importa: primeiro checa login, depois o grupo
            return [IsAuthenticated(), IsGerente()]

        # Para GET, PUT, PATCH: Basta estar logado (e ser dono, garantido pelo queryset)
        return [IsAuthenticated()]
```

Capítulo 6: Testando a Segurança Completa

Como nas apostilas anteriores, vamos validar cada passo.

6.1. Teste de Cadastro e Cargo

1. **Requisição:** POST /api/register/
   - **JSON:** {"username": "funcionario", "password": "123", "email": "func@empresa.com"}
2. **Resultado Esperado:** 201 Created.
3. **Verificação:** Vá ao Django Admin (/admin), localize o usuário funcionario e verifique se ele está dentro do grupo **Comum**.

6.2. Teste de Isolamento (Row Level Security)

1. Logue com admin e crie uma Tarefa "Tarefa Secreta do Chefe".
2. Logue com funcionario (pegue o token JWT).
3. Faça GET /api/tarefas/ usando o token do funcionário.
4. **Resultado Esperado:** A lista deve voltar vazia [] ou apenas com as tarefas criadas pelo

funcionário. A "Tarefa Secreta" **não** deve aparecer.

6.3. Teste de RBAC (Bloqueio de Delete)

1. Crie uma tarefa usando o token do funcionario. Anote o ID (ex: 10).
2. Tente DELETE /api/tarefas/10/ com o token do funcionario.
3. **Resultado Esperado:** 403 Forbidden
    - **Body:** {"detail": "Você não tem permissão para executar essa ação."}.
    - *Motivo:* O usuário está no grupo "Comum", e a View exige "Gerente" para DELETE.

6.4. Teste de RBAC (Sucesso de Delete)

1. Acesse o Django Admin como superusuário.
2. Edite o usuário funcionario e adicione-o ao grupo **Gerente**. Salve.
3. Repita a requisição DELETE /api/tarefas/10/.
4. **Resultado Esperado:** 204 No Content (Sucesso).

Resumo da Apostila

O que aprendemos e implementamos:

1. **Fim da exposição de dados:** Substituímos .all() por .filter(user=request.user) no get_queryset, garantindo privacidade absoluta entre usuários.
2. **Segurança por Omissão:** Aprendemos que, ao filtrar o queryset, acessos indevidos retornam 404 (dado não encontrado) em vez de 403 (acesso negado), o que é mais seguro.
3. **Cadastro Seguro:** Implementamos registro público com hashing de senha e ocultação do campo senha na resposta.
4. **Automação de Papéis:** Todo usuário novo nasce automaticamente com o cargo "Comum" graças à lógica no Serializer.
5. **Poderes Diferenciados:** Implementamos a lógica onde apenas usuários com o cargo "Gerente" podem destruir dados.

Exercícios Práticos

Para consolidar o conhecimento, realize os 3 exercícios abaixo. Eles simulam demandas reais de mercado.

Exercício 1: Bloqueio de Alteração de E-mail

Em muitos sistemas, o usuário não pode alterar seu e-mail após o cadastro (pois é sua chave primária de login), ou precisa de um fluxo especial.

- **Objetivo:** Modifique o UserSerializer (ou crie um novo UserUpdateSerializer) usado para PUT/PATCH de usuários.
- **Regra:** O campo email deve ser **Read Only** durante atualizações. O usuário pode mudar o nome, mas não o email.
- **Dica:** Use read_only_fields na classe Meta.

Exercício 2: Permissão "IsAdminOrOwner"

Crie uma permissão customizada mais flexível em core/permissions.py.

- **Cenário:** Às vezes, o suporte técnico (Admin/Staff) precisa acessar a tarefa de um usuário para debug, mas o get_queryset atual impede isso (retorna 404 para o admin).
- **Tarefa A (Permissão):** Crie a classe IsAdminOrOwner. Ela retorna True se request.user.is_staff for verdadeiro.
- **Tarefa B (Queryset):** Modifique o get_queryset na View. Se o usuário for is_staff, retorne Tarefa.objects.all(). Se não, retorne filter(user=user).
- **Resultado:** O Admin deve conseguir ver todas as tarefas; usuários comuns apenas as suas.

Exercício 3: Endpoint de "Meus Dados" (/api/me/)

Crie um endpoint que retorne os dados do próprio usuário logado (id, username, email, grupos).

- **View:** Use RetrieveAPIView.
- **Objeto:** O método get_object deve retornar simplesmente self.request.user.
- **Serializer:** Crie um UserProfileSerializer que inclua um campo cargo (que exibe o nome do grupo do usuário, ex: "Gerente").
- **Dica:** Use StringRelatedField ou SlugRelatedField para mostrar o nome do grupo em vez do ID.

Próximo Passo: Com a segurança robusta implementada, sua API está pronta para produção. Na **Apostila 6**, aprenderemos a garantir que tudo isso continue funcionando no futuro através de **Testes Automatizados (Unitários e de Integração)**.