Apostila 4: JWT, Autenticação e Proteção Completa de Rotas (Revisada)

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

🎯 Objetivos de Aprendizagem

Ao final desta apostila, você será capaz de:

- Compreender o que é **JWT** e como funciona.
- Configurar a autenticação **JWT no Django REST Framework**.
- Proteger rotas com IsAuthenticated.
- **Injetar automaticamente o usuário logado** nas operações (perform_create).
- Implementar *login*, *logout* e **renovação de tokens**.
- Resolver o problema da **ForeignKey obrigatória** (user não pode ser nulo).
- Criar uma API totalmente segura.

Capítulo 1: O Problema da Segurança 🔐

1.1. Recapitulando: O Campo user Temporário

A abordagem anterior, tornando o campo user opcional, gerava graves problemas:

Python

```
# core/models.py (VERSÃO TEMPORÁRIA - INSEGURA)
class Tarefa(models.Model):
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        null=True,      # ❌ TEMPORÁRIO
        blank=True      # ❌ TEMPORÁRIO
    )
    # ... outros campos
```

Problemas desta abordagem:

- ❌ **Tarefas órfãs:** Tarefas sem dono no banco.
- ❌ **Segurança zero:** Qualquer um pode acessar/modificar qualquer tarefa.
- ❌ **Violação de Integridade:** O banco de dados fica inconsistente.

1.2. A Solução Completa: Autenticação JWT

O **JWT (JSON Web Token)** é um padrão compacto e seguro para a transmissão de informações entre partes como um objeto JSON. Ele permite que o servidor verifique a autenticidade dos dados sem a necessidade de manter o estado (sem estado - *stateless*).

FLUXO DE AUTENTICAÇÃO JWT

1. **Cliente** faz **LOGIN** (POST /api/token/) com username e password.
2. **Servidor** valida as credenciais e retorna um **Access Token** e um **Refresh Token**.
3. **Cliente** guarda os tokens.
4. **Cliente** usa o Access Token no *header* Authorization: Bearer <TOKEN> em **TODAS** as requisições protegidas.
5. **Servidor** valida o token e decodifica o payload para identificar o usuário (request.user).
6. **View** usa request.user para executar a lógica de negócio (ex: serializer.save(user=request.user)).

1.3. Por Que JWT?

Característica	Sessões (Cookies)	JWT (Token)
Storage	Servidor guarda estado	Cliente guarda token
Escalabilidade	Difícil (precisa replicar sessões)	**Fácil (stateless)**
Mobile	Difícil (cookies)	**Fácil (header)**
CORS	Problemático	**Simples**
Expiração	Servidor controla	Token tem **TTL (Time To Live)**

Capítulo 2: Anatomia do JWT 🔬

2.1. Estrutura de um Token JWT

Um JWT tem 3 partes separadas por pontos (.): **HEADER.PAYLOAD.SIGNATURE**.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxLCJ1c2VybmFtZSI6ImpvYW8iLCJleHAiOjE2MzI1MDAwMDB9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Parte 1: HEADER (Cabeçalho)

Especifica o tipo de token (JWT) e o algoritmo de criptografia (alg: HS256).

Parte 2: PAYLOAD (Dados/Claims)

Contém as "claims" (informações) sobre o usuário e metadados do token.
Exemplo: user_id, username, exp (expiração), iat (criado em).

Parte 3: SIGNATURE (Assinatura)

Gerada com uma chave secreta (SECRET_KEY) que **só o servidor conhece**.

$$\text{Signature} = \text{HMACSHA256}(\text{Base64UrlEncode(Header) + "." + Base64UrlEncode(Payload)}, \text{SECRET\_KEY})$$

A assinatura garante a **integridade** (não foi modificado) e a **autenticidade** (foi gerado pelo nosso servidor) do token.

2.2. Access Token vs Refresh Token

Token	Validade	Propósito	Quando usar
Access Token	**Curta (5-15 min)**	Acessar recursos protegidos	Em toda requisição da API
Refresh Token	**Longa (1-7 dias)**	Renovar Access Token	Quando o Access Token expira

O uso do Refresh Token é a chave para a segurança e usabilidade: ele permite que o Access Token expire rapidamente (segurança), mas evita que o usuário tenha que logar a cada 5 minutos (usabilidade).

Capítulo 3: Configuração JWT no Django 🛠️

3.1. Instalação e Configurar settings.py

Já instalamos o djangorestframework-simplejwt anteriormente. Agora, vamos configurá-lo.

```
# config/settings.py
from datetime import timedelta

# ... (imports e configurações existentes) ...

INSTALLED_APPS = [
    # ... (apps existentes)
    'rest_framework',
    'rest_framework_simplejwt',  # ← JWT
    # ...
]

# Configuração do Django REST Framework
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        # Define JWT como método de autenticação PADRÃO
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    # Outras configurações padrão
}

# Configuração do Simple JWT
SIMPLE_JWT = {
    # Tempo de vida do Access Token (curto!)
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),

    # Tempo de vida do Refresh Token (longo!)
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),

    # Define o esquema de autenticação no header HTTP
    'AUTH_HEADER_TYPES': ('Bearer',),

    # Algoritmo de criptografia
    'ALGORITHM': 'HS256',

    # Nome do campo de usuário no payload (user_id é padrão)
    'USER_ID_CLAIM': 'user_id',
}
```

3.2. Configurar URLs

Edite config/urls.py para incluir os *endpoints* de autenticação e renovação.

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include
from rest_framework_simplejwt.views import (
    TokenObtainPairView,    # Login (obter access e refresh tokens)
    TokenRefreshView,       # Renovar token
)

urlpatterns = [
    path('admin/', admin.site.urls),

    # JWT: Endpoints de autenticação
    path('api/token/',
         TokenObtainPairView.as_view(),
         name='token_obtain_pair'),

    path('api/token/refresh/',
         TokenRefreshView.as_view(),
         name='token_refresh'),

    # App core
    path('api/', include('core.urls')),
]
```

Capítulo 4: Protegendo as Views 🛡️

4.1. A Permissão IsAuthenticated

A permissão IsAuthenticated verifica se a requisição foi autenticada, ou seja, se um token JWT válido foi fornecido e decodificado, resultando em um objeto User em request.user.

```
from rest_framework.permissions import IsAuthenticated

class MinhaView(APIView):
    # Adicionando a permissão
    permission_classes = [IsAuthenticated]

    def get(self, request):
        # Se chegou aqui, request.user é SEMPRE um objeto User logado
        print(f"Usuário autenticado: {request.user.username}")
        # ...
```

4.2. Usando generics.ListCreateAPIView (Melhor Prática DRF) 🏆

Em vez de usar APIView e reescrever o método post, é altamente recomendado usar as classes genéricas do DRF. Para Listar e Criar recursos, usamos ListCreateAPIView.

Em core/views.py:

```
# 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 tarefas e permite a criação de novas tarefas.

    PROTEGIDA: Requer autenticação JWT.
    """
    queryset = Tarefa.objects.all()
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated] # ← Proteção

    # MÉTODO CHAVE: Injeta o usuário logado antes de salvar o objeto
    def perform_create(self, serializer):
```

```
        """
        Associa a tarefa ao usuário logado (request.user) automaticamente.
        """
        # request.user é garantido como autenticado pelo IsAuthenticated
        serializer.save(user=self.request.user)

# A URL deve apontar para esta view em core/urls.py
# path('tarefas/', TarefaListCreateAPIView.as_view(), name='tarefa-list-create'),
```

4.3. Protegendo View de Detalhe

Para operações em um único recurso (GET, PUT, PATCH, DELETE), usamos generics.RetrieveUpdateDestroyAPIView.

Em core/views.py:

```
# core/views.py (continuação)

class TarefaRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):
    """
    Detalhes de tarefa, atualização e exclusão.

    PROTEGIDA: Requer autenticação JWT.
    """
    queryset = Tarefa.objects.all()
    serializer_class = TarefaSerializer
    permission_classes = [IsAuthenticated] # ← Proteção

# A URL deve apontar para esta view em core/urls.py
# path('tarefas/<int:pk>/', TarefaRetrieveUpdateDestroyAPIView.as_view(), name='tarefa-detail'),
```

Capítulo 5: Ajustando Model e Serializer 🔧

5.1. Reverter Model para Seguro (user Obrigatório)

Em core/models.py, remova os parâmetros null=True e blank=True.

```
# core/models.py (VERSÃO FINAL - SEGURA)
from django.db import models
from django.contrib.auth.models import User

class Tarefa(models.Model):
    """
    Model de Tarefa com segurança. O campo 'user' é OBRIGATÓRIO.
    """
    # ↓ OBRIGATÓRIO: Sem null=True e sem blank=True
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name='tarefas',
        verbose_name='Usuário'
    )
    # ... (outros campos)
```

Criar Migração:

```
python manage.py makemigrations core
# Se houver tarefas sem user, será solicitado um valor padrão
# Sugestão: Digite o ID do seu superusuário para migrar tarefas órfãs

python manage.py migrate
```

5.2. Ajustar Serializer (Ocultar user na Entrada)

Em core/serializers.py, o campo user deve ser apenas de leitura na saída (para mostrar o nome do dono), mas não deve ser aceito na entrada.

```
# core/serializers.py (VERSÃO FINAL)
from rest_framework import serializers
from .models import Tarefa

class TarefaSerializer(serializers.ModelSerializer):
    """
    Serializer para Tarefa com segurança.
    O campo 'user' é exibido (read-only) mas NÃO aceito na entrada.
    """

    # 1. Mostra o username do usuário em vez do ID (read-only na saída)
    user = serializers.StringRelatedField(read_only=True)

    class Meta:
        model = Tarefa
        fields = ['id', 'user', 'titulo', 'concluida', 'criada_em']

        # 2. Impedir que o cliente envie/edite esses campos
        read_only_fields = ['id', 'user', 'criada_em']
```

Com esta configuração, o JSON de entrada para o POST será simplificado:

```
// Entrada (POST):
{
  "titulo": "Nova tarefa",
  "concluida": false
}
// O servidor injeta 'user' automaticamente via perform_create
```

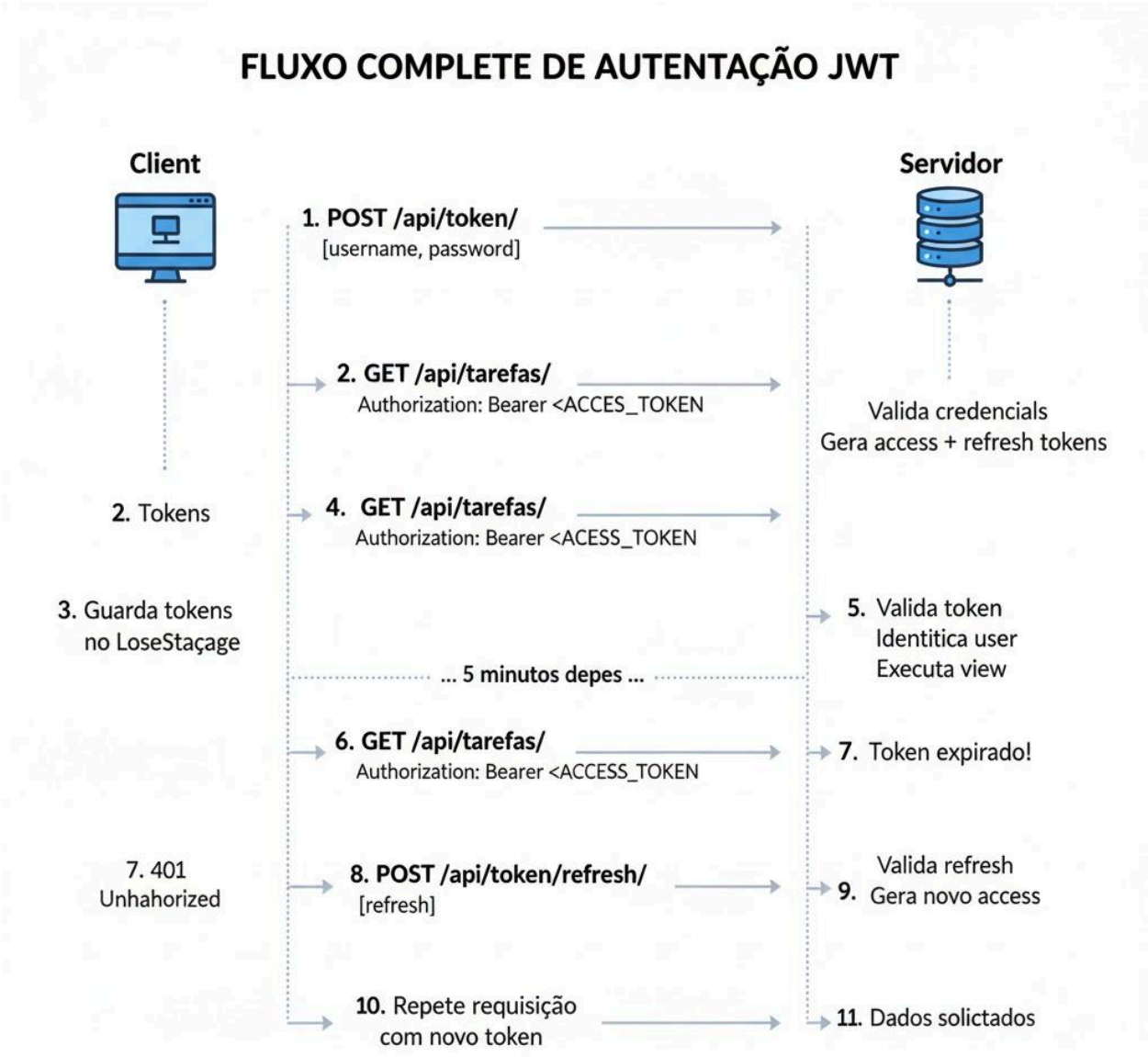
Capítulo 6: Testando Autenticação Completa 🧪

A seção de testes detalhada na apostila original é excelente. Ela cobre os cenários críticos:

1. **Tentar Acessar Sem Token:** Deve retornar 401 Unauthorized.
2. **Fazer Login:** POST /api/token/ deve retornar access e refresh tokens.
3. **Criar Tarefa Com Token:** POST /api/tarefas/ com Authorization: Bearer <ACCESS_TOKEN> deve criar a tarefa e injetar o user correto.
4. **Listar Tarefas Com Token:** GET /api/tarefas/ deve funcionar.
5. **Token Expirado:** Deve retornar 401 Unauthorized com a mensagem "Token is expired".
6. **Renovar Token:** POST /api/token/refresh/ com o refresh token deve gerar um novo access token.
7. **Login com Credenciais Erradas:** Deve retornar 401 Unauthorized.

Capítulo 7: Fluxo Completo de Autenticação 🔄

O diagrama de sequência é fundamental para entender o ciclo de vida completo do token.



Capítulo 8: Customizando e Gerenciando o Token JWT 🎨

8.1. Adicionar Campos Extras no Token (Custom Claims)

Para incluir dados adicionais do usuário (como email ou is_staff) diretamente no *payload* do token, você precisa customizar o TokenObtainPairSerializer.

```
# core/serializers.py (adicionar)
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer

class CustomTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        # Adicionar campos customizados ao payload
        token['username'] = user.username
        token['email'] = user.email
        token['is_staff'] = user.is_staff

        return token
```

Em seguida, use esta classe em uma *View* customizada:

```
# core/views.py (adicionar no final)
from rest_framework_simplejwt.views import TokenObtainPairView
from .serializers import CustomTokenObtainPairSerializer

class CustomTokenObtainPairView(TokenObtainPairView):
    """View que usa o serializer customizado."""
    serializer_class = CustomTokenObtainPairSerializer
```

Atualize config/urls.py para usar a view customizada:

```
# config/urls.py
from core.views import CustomTokenObtainPairView  # Import customizado
# ...
urlpatterns = [
    # ...
    path('api/token/',
         CustomTokenObtainPairView.as_view(),  # ← View customizada
         name='token_obtain_pair'),
    # ...
]
```

8.2. Endpoint para Logout (Blacklist)

O logout em JWT significa tornar o Refresh Token (e, opcionalmente, o Access Token) inválido antes de sua expiração natural. Isso é feito adicionando-o a uma lista negra (*blacklist*).

1. **Instalar e Configurar:**
   Bash
   pip install djangorestframework-simplejwt[crypto]

   Em settings.py:
   Python
   INSTALLED_APPS = [
       # ...
       'rest_framework_simplejwt.token_blacklist', # ← Adicionar
   ]
   # Configure SIMPLE_JWT para rotação e blacklist se necessário (ver Cap. 9)

   # Rodar migração
   python manage.py migrate

2. **View de Logout** (em core/views.py):

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from rest_framework.permissions import IsAuthenticated
from rest_framework_simplejwt.tokens import RefreshToken

class LogoutView(APIView):
    permission_classes = [IsAuthenticated]

    def post(self, request):
        try:
            refresh_token = request.data.get("refresh")
            token = RefreshToken(refresh_token)
            token.blacklist()  # Adiciona o token à lista negra

            return Response(
                {"detail": "Logout realizado com sucesso."},
                status=status.HTTP_205_RESET_CONTENT # 205 é a resposta padrão para "reset content"
            )
        except Exception: # Captura exceções como token_not_valid
            return Response(
                {"detail": "Token inválido."},
                status=status.HTTP_400_BAD_REQUEST
            )
```

3. **Adicionar URL** (em core/urls.py):

```
# core/urls.py (no seu app 'core')
from django.urls import path
from .views import LogoutView, TarefaListCreateAPIView, TarefaRetrieveUpdateDestroyAPIView

urlpatterns = [
    path('tarefas/', TarefaListCreateAPIView.as_view(), name='tarefa-list-create'),
    path('tarefas/<int:pk>/', TarefaRetrieveUpdateDestroyAPIView.as_view(), name='tarefa-detail'),
    path('logout/', LogoutView.as_view(), name='logout'), # ← Novo endpoint
]
```

Capítulo 9: Segurança Avançada 🔐

9.1. Configurações de Segurança em Produção

Configurações mais robustas em config/settings.py:

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),  # Curto e suficiente
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),

    # Rotacionar refresh tokens (Melhor Prática de Segurança!)
    # A cada renovação, o token antigo é invalidado e um novo é gerado.
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,  # Requer 'token_blacklist'

    # ... (outras configurações como ALGORITHM: 'HS256')
}
```

9.2. Rate Limiting (Throttling)

Para prevenir ataques de força bruta ou abuso de API, configure o *Rate Limiting* (ou *Throttling*) no DRF em config/settings.py.

```
REST_FRAMEWORK = {
    # ... (authentication classes)

    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',  # 100 requisições por dia para anônimos (ex: /token/)
        'user': '3000/day'  # 3000 requisições por dia para autenticados (ex: /tarefas/)
    }
}
```

🎓 Resumo da Apostila

O que aprendemos:

- ✅ Conceito e estrutura de **JWT** (header.payload.signature).
- ✅ Configuração completa de **Simple JWT** no Django.
- ✅ Proteção de rotas com permission_classes = [IsAuthenticated].
- ✅ **Injeção automática do usuário** usando perform_create(serializer.save(user=self.request.user)).
- ✅ Fluxo de **login** (/api/token/), **renovação** (/api/token/refresh/) e **logout** (/api/logout/).
- ✅ Restabelecemos a **integridade do Model** tornando o campo user obrigatório.
- ✅ Melhores práticas do DRF usando generics.ListCreateAPIView.

📝 Exercícios Práticos

Exercício 1: Endpoint de Profile

Crie endpoint `/api/me/` que retorna dados do usuário autenticado:

```
class MeView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        user = request.user
        return Response({
            'id': user.id,
            'username': user.username,
            'email': user.email,
            'is_staff': user.is_staff,
            'date_joined': user.date_joined
        })
```

Exercício 2: Mudança de Senha

Crie endpoint `/api/change-password/` protegido:

```
class ChangePasswordView(APIView):
    permission_classes = [IsAuthenticated]

    def post(self, request):
        user = request.user
        old_password = request.data.get('old_password')
        new_password = request.data.get('new_password')

        if not user.check_password(old_password):
            return Response(
                {'error': 'Senha atual incorreta'},
                status=400
            )

        user.set_password(new_password)
        user.save()

        return Response({'detail': 'Senha alterada com sucesso'})
```

Exercício 3: Estatísticas do Usuário

Crie endpoint `/api/stats/` que retorna:

```
{
  "total_tarefas": 10,
  "concluidas": 6,
  "pendentes": 4,
  "taxa_conclusao": 0.6
}
```

🔍 Checklist de Verificação

Antes de prosseguir para a Apostila 5:

- [ ] JWT configurado em settings.py com ACCESS_TOKEN_LIFETIME curto.
- [ ] Endpoints /api/token/ e /api/token/refresh/ funcionando.
- [ ] Consegue fazer login e obter tokens.
- [ ] Views **principais** usam permission_classes = [IsAuthenticated].
- [ ] TarefaListCreateAPIView usa perform_create para injetar o usuário.
- [ ] Campo user voltou a ser **obrigatório** no Model (null=False, blank=False).
- [ ] TarefaSerializer tem user como serializers.StringRelatedField(read_only=True).
- [ ] Entende diferença entre **access** (curto) e **refresh** (longo) token.

🎉 Parabéns! Sua API agora está **completamente autenticada e segura!**

Na próxima apostila, vamos implementar **Autorização** (Capítulo 11) para garantir que cada usuário **só possa ver/editar suas próprias tarefas**, protegendo contra o acesso cruzado de dados.