

# Apostila 2: CRUD - Criação de Recursos e Validação Avançada (POST)

## 🎯 Objetivos de Aprendizagem

Ao final desta apostila, você será capaz de:

- Implementar o método POST para criação de recursos
- Entender e aplicar validação de dados com Serializers
- Trabalhar com status codes HTTP adequados
- Implementar validações customizadas
- Tratar erros de forma profissional
- Criar uma API completa de Leitura e Criação

## Capítulo 1: Revisão e O Verbo POST



### 1.1. Recapitulando a Apostila 1

Na apostila anterior, implementamos:

```
# ✅ Model: Estrutura dos dados
class Tarefa(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    titulo = models.CharField(max_length=200)
    concluida = models.BooleanField(default=False)
    criada_em = models.DateTimeField(auto_now_add=True)

# ✅ Serializer: Tradução Python ↔ JSON
class TarefaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarefa
        fields = ['id', 'user', 'titulo', 'concluida', 'criada_em']

# ✅ View: Lógica do GET
class ListaTarefasAPIView(APIView):
    def get(self, request, format=None):
        tarefas = Tarefa.objects.all()
        serializer = TarefaSerializer(tarefas, many=True)
        return Response(serializer.data)

# ✅ URL: /api/tarefas/ → ListaTarefasAPIView
```

**Resultado:** Conseguimos **LISTAR** tarefas. Agora vamos **CRIAR**.

## 1.2. O Verbo POST: Anatomia de uma Requisição de Criação

### GET vs POST: A Diferença Fundamental

Característica	GET	POST
<b>Propósito</b>	Ler dados	Criar novo recurso
<b>Body</b>	Não tem	Tem (JSON com dados)
<b>Idempotência</b>	Sim (repetir não muda nada)	Não (cada vez cria um novo)
<b>Cache</b>	Pode ser cacheado	Nunca é cacheado
<b>Segurança</b>	Seguro (não modifica)	Não seguro (modifica estado)

### Estrutura de uma Requisição POST

```
POST /api/tarefas/ HTTP/1.1
Host: localhost:8000
Content-Type: application/json
Content-Length: 58
```

```
{
  "titulo": "Implementar método POST",
  "concluida": false
}
```

### Componentes:

- Método:** POST (indica criação)
- URL:** /api/tarefas/ (endpoint de coleção)
- Content-Type:** application/json (formato dos dados)
- Body:** JSON com os dados da nova tarefa

### Resposta de Sucesso

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/tarefas/42/
```

```
{
  "id": 42,
  "user": 1,
  "titulo": "Implementar método POST",
  "concluida": false,
  "criada_em": "2025-11-29T14:30:00Z"
}
```

### Detalhes importantes:

- Status 201 Created (não 200 OK)
- Header Location aponta para o recurso criado
- Body contém o objeto completo (com id gerado)

## 1.3. O Desafio da Validação

Por que validar?

```
# ❌ Sem validação, isso poderia ser salvo:  
{  
    "titulo": "", # Título vazio  
    "concluida": "talvez", # Não é booleano  
    "user": 9999 # Usuário inexistente  
}
```

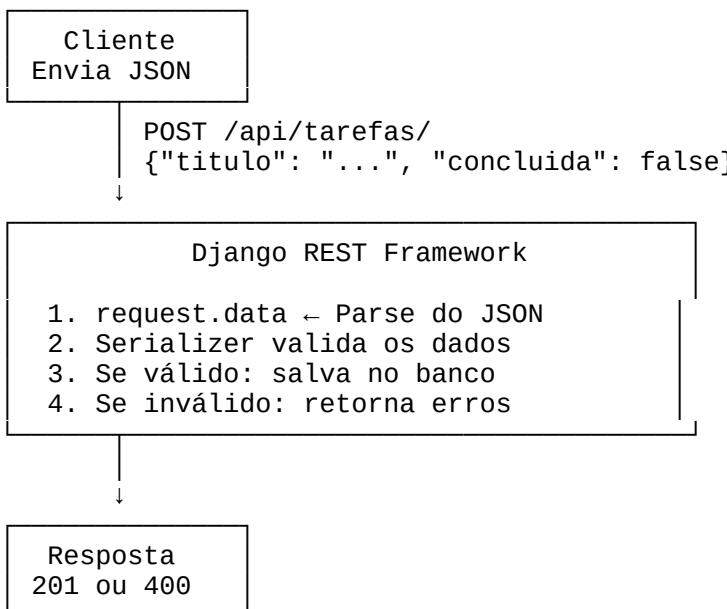
Consequências:

- Dados inconsistentes no banco
- Bugs difíceis de rastrear
- Má experiência do usuário
- Vulnerabilidades de segurança

O Serializer do DRF resolve isso automaticamente!

## Capítulo 2: Implementando o Método POST

### 2.1. O Fluxo Completo do POST



## 2.2. Código: Implementação do POST

Edite `core/views.py` e adicione o método `post`:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Tarefa
from .serializers import TarefaSerializer

class ListaTarefasAPIView(APIView):
    """
    View para operações de coleção (GET e POST).

    Endpoints:
        GET /api/tarefas/ - Lista todas as tarefas
        POST /api/tarefas/ - Cria uma nova tarefa
    """

    def get(self, request, format=None):
        """Lista todas as tarefas."""
        tarefas = Tarefa.objects.all()
        serializer = TarefaSerializer(tarefas, many=True)
        return Response(serializer.data, status=status.HTTP_200_OK)

    def post(self, request, format=None):
        """
        Cria uma nova tarefa.

        Args:
            request.data: JSON com dados da tarefa
            {
                "titulo": "string",
                "concluida": boolean (opcional, default=False)
            }

        Returns:
            201 Created: Tarefa criada com sucesso
            400 Bad Request: Dados inválidos
        """
        # 1. INSTANCIAR: Criar serializer com dados recebidos
        serializer = TarefaSerializer(data=request.data)

        # 2. VALIDAR: Checar se os dados são válidos
        if serializer.is_valid():
            # 3. SALVAR: Persistir no banco de dados
            serializer.save()

            # 4. RESPONDER: Retornar objeto criado + status 201
            return Response(
                serializer.data,
                status=status.HTTP_201_CREATED
            )

        # 5. ERRO: Retornar erros de validação + status 400
        return Response(
            serializer.errors,
            status=status.HTTP_400_BAD_REQUEST
        )
```

## 2.3. Análise Linha por Linha

### Linha 1: Instanciar o Serializer

```
serializer = TarefaSerializer(data=request.data)
```

#### O que acontece aqui:

- `request.data`: DRF já parseou o JSON automaticamente
- `data=request.data`: Passamos os dados para validação
- **Importante:** Não salvamos ainda! Apenas preparamos.

### Linha 2: Validação

```
if serializer.is_valid():
```

#### O que `is_valid()` faz:

##### 1. Validação de tipo:

```
"concluida": "sim" # ❌ Erro: esperado boolean
```

##### 2. Validação de tamanho:

```
"titulo": "a" * 300 # ❌ Erro: max_length=200
```

##### 3. Validação de obrigatoriedade:

```
{# ❌ Erro: 'titulo' é obrigatório
```

##### 4. Validação de ForeignKey:

```
"user": 9999 # ❌ Erro: usuário não existe
```

### Linha 3: Salvar

```
serializer.save()
```

#### O que acontece:

- Cria instância do Model
- Executa `INSERT` no banco
- Atribui `id` gerado automaticamente
- Preenche campos auto (como `criada_em`)

### Linha 4: Resposta de Sucesso

```
return Response(serializer.data, status=status.HTTP_201_CREATED)
```

#### Por que 201 Created?

- Padrão REST para criação bem-sucedida
- Indica que um novo recurso foi criado
- Diferencia de 200 OK (operação genérica)

### Linha 5: Resposta de Erro

```
return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

### Formato de `serializer.errors`:

```
{  
    "titulo": [  
        "Este campo é obrigatório."  
    ],  
    "concluida": [  
        "Insira um valor booleano válido."  
    ]  
}
```

## 2.4. Status Codes: A Comunicação Correta

Situação	Status	Motivo
Tarefa criada	201 Created	Novo recurso foi criado
Dados inválidos	400 Bad Request	Erro do cliente (dados ruins)
Não autenticado	401 Unauthorized	Falta de autenticação
Sem permissão	403 Forbidden	Autenticado, mas sem acesso
Servidor quebrou	500 Server Error	Bug no código

### Uso correto:

```
# ✅ CERTO  
return Response(data, status=status.HTTP_201_CREATED)  
  
# ❌ ERRADO (usar 200 para criação)  
return Response(data, status=status.HTTP_200_OK)
```

## Capítulo 3: O Desafio da ForeignKey (user)

### 3.1. O Problema

Se você tentar criar uma tarefa agora:

```
POST /api/tarefas/  
{  
    "titulo": "Minha tarefa",  
    "concluida": false  
}
```

### Erro:

```
{  
    "user": [  
        "Este campo é obrigatório."  
    ]  
}
```

### Por quê?

- O Model exige `user` (ForeignKey sem `null=True`)
- O Serializer valida baseado no Model
- O cliente não pode enviar `user` (seria inseguro!)

### 3.2. Solução Temporária: Tornar user Opcional

Para fins didáticos, vamos tornar o campo opcional. Na Apostila 4, implementaremos JWT e injeção automática do usuário.

Edite core/models.py:

```
from django.db import models
from django.contrib.auth.models import User

class Tarefa(models.Model):
    # MODIFICAÇÃO TEMPORÁRIA: null=True, blank=True
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        null=True,          # Permite NULL no banco
        blank=True,         # Permite vazio em formulários
        related_name='tarefas',
        verbose_name='Usuário'
    )

    titulo = models.CharField(max_length=200, verbose_name='Título')
    concluida = models.BooleanField(default=False, verbose_name='Concluída')
    criada_em = models.DateTimeField(auto_now_add=True, verbose_name='Criada em')

    class Meta:
        verbose_name = 'Tarefa'
        verbose_name_plural = 'Tarefas'
        ordering = ['-criada_em']

    def __str__(self):
        return f"{self.titulo} ({'✓' if self.concluida else '✗'})"
```

**Criar e aplicar migração:**

```
python manage.py makemigrations core
# Será perguntado: "Provide a one-off default now"
# Digite: 1 (para usar None)

python manage.py migrate
```

### 3.3. Ajustar o Serializer

Remova temporariamente `user` dos campos obrigatórios.

Edite `core/serializers.py`:

```
from rest_framework import serializers
from .models import Tarefa

class TarefaSerializer(serializers.ModelSerializer):
    """
    Serializer para Tarefa.

    Nota: O campo 'user' foi removido temporariamente.
    Na Apostila 4, será injetado automaticamente pelo servidor.
    """

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

**Agora funciona:**

```
POST /api/tarefas/
{
    "titulo": "Minha primeira tarefa via API",
    "concluida": false
}

→ 201 Created
{
    "id": 1,
    "titulo": "Minha primeira tarefa via API",
    "concluida": false,
    "criada_em": "2025-11-29T15:00:00Z"
}
```

## Capítulo 4: Validação Avançada

### 4.1. Validações Nativas do DRF

O Serializer herda validações do Model:

#### `max_length (CharField)`

```
# Model
titulo = models.CharField(max_length=200)

# Teste
POST /api/tarefas/
{"titulo": "a" * 300} # 300 caracteres

# Resposta: 400 Bad Request
{
    "titulo": [
        "Certifique-se de que este campo não tenha mais de 200 caracteres."
    ]
}
```

## Campo obrigatório

```
POST /api/tarefas/
{"concluida": true} # Faltou 'titulo'

# Resposta: 400
{
  "titulo": [
    "Este campo é obrigatório."
  ]
}
```

## Tipo incorreto

```
POST /api/tarefas/
{
  "titulo": "Tarefa",
  "concluida": "sim" # String em vez de boolean
}

# Resposta: 400
{
  "concluida": [
    "Insira um valor booleano válido."
  ]
}
```

## 4.2. Validações Customizadas: validate\_<campo>

Você pode adicionar validações específicas no Serializer:

```
from rest_framework import serializers
from .models import Tarefa

class TarefaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarefa
        fields = ['id', 'titulo', 'concluida', 'criada_em']
        read_only_fields = ['id', 'criada_em']

    def validate_titulo(self, value):
        """
        Validação customizada para o campo 'titulo'.

        Regras:
        - Não pode ser vazio (após strip)
        - Não pode conter apenas números
        - Deve ter pelo menos 3 caracteres
        """
        # Remover espaços em branco
        value = value.strip()

        # Validação 1: Não vazio
        if not value:
            raise serializers.ValidationError(
                "O título não pode ser vazio ou conter apenas espaços."
            )

        # Validação 2: Mínimo de caracteres
        if len(value) < 3:
            raise serializers.ValidationError(
                "O título deve ter pelo menos 3 caracteres."
            )
```

```

# Validação 3: Não apenas números
if value.isdigit():
    raise serializers.ValidationError(
        "O título não pode conter apenas números."
    )

return value

```

### Testes:

```

# ❌ Título muito curto
POST {"titulo": "ab"}
→ 400: "O título deve ter pelo menos 3 caracteres."

# ❌ Apenas números
POST {"titulo": "123"}
→ 400: "O título não pode conter apenas números."

# ❌ Espaços vazios
POST {"titulo": "   "}
→ 400: "O título não pode ser vazio..."

# ✅ Válido
POST {"titulo": "Minha tarefa"}
→ 201 Created

```

## 4.3. Validação de Múltiplos Campos: validate()

Para validar relações entre campos:

```

class TarefaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarefa
        fields = ['id', 'titulo', 'concluida', 'criada_em']
        read_only_fields = ['id', 'criada_em']

    def validate(self, data):
        """
        Validação de objeto completo (múltiplos campos).

        Exemplo: Tarefas com palavra "urgente" não podem
        começar como concluídas.
        """
        titulo = data.get('titulo', '').lower()
        concluida = data.get('concluida', False)

        if 'urgente' in titulo and concluida:
            raise serializers.ValidationError(
                "Tarefas urgentes não podem ser criadas como concluídas."
            )

        return data

```

### Teste:

```
POST /api/tarefas/
{
    "titulo": "URGENTE: Resolver bug",
    "concluida": true
}

→ 400 Bad Request
{
    "non_field_errors": [
        "Tarefas urgentes não podem ser criadas como concluídas."
    ]
}
```

## 4.4. Validações Assíncronas: Unicidade

Validar se algo já existe no banco:

```
class TarefaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarefa
        fields = ['id', 'titulo', 'concluida', 'criada_em']
        read_only_fields = ['id', 'criada_em']

    def validate_titulo(self, value):
        """Impedir títulos duplicados para o mesmo usuário."""
        user = self.context['request'].user

        if Tarefa.objects.filter(user=user, titulo=value).exists():
            raise serializers.ValidationError(
                "Você já tem uma tarefa com este título."
            )

    return value
```

**Importante:** Precisa passar `context` na View:

```
def post(self, request, format=None):
    serializer = TarefaSerializer(
        data=request.data,
        context={'request': request} # Passa o request
    )
    # ...
```

# Capítulo 5: Tratamento de Erros Profissional

## 5.1. Estrutura de Resposta de Erro

**Padrão do DRF:**

```
{
    "field_name": [
        "Mensagem de erro 1",
        "Mensagem de erro 2"
    ],
    "another_field": [
        "Outro erro"
    ]
}
```

## **Erros não relacionados a campo específico:**

```
{  
    "non_field_errors": [  
        "Erro geral da validação"  
    ]  
}
```

## **5.2. Customizar Mensagens de Erro**

```
class TarefaSerializer(serializers.ModelSerializer):  
    # Customizar mensagens padrão  
    titulo = serializers.CharField(  
        max_length=200,  
        error_messages={  
            'required': 'O título é obrigatório.',  
            'blank': 'O título não pode ser vazio.',  
            'max_length': 'O título não pode ter mais de 200 caracteres.'  
        }  
    )  
  
    class Meta:  
        model = Tarefa  
        fields = ['id', 'titulo', 'concluida', 'criada_em']  
        read_only_fields = ['id', 'criada_em']
```

## **5.3. Tratamento de Exceções na View**

```
from rest_framework.exceptions import ValidationError  
from django.db import IntegrityError  
  
class ListaTarefasAPIView(APIView):  
    def post(self, request, format=None):  
        try:  
            serializer = TarefaSerializer(data=request.data)  
  
            if serializer.is_valid():  
                serializer.save()  
                return Response(  
                    serializer.data,  
                    status=status.HTTP_201_CREATED  
                )  
  
            return Response(  
                serializer.errors,  
                status=status.HTTP_400_BAD_REQUEST  
            )  
  
        except IntegrityError as e:  
            # Erro de constraint no banco (ex: UNIQUE)  
            return Response(  
                {'error': 'Violação de integridade no banco de dados.'},  
                status=status.HTTP_400_BAD_REQUEST  
            )  
  
        except Exception as e:  
            # Erro inesperado  
            return Response(  
                {'error': 'Erro interno do servidor.'},  
                status=status.HTTP_500_INTERNAL_SERVER_ERROR  
            )
```

## 5.4. Logging de Erros

```
import logging

logger = logging.getLogger(__name__)

class ListaTarefasAPIView(APIView):
    def post(self, request, format=None):
        try:
            serializer = TarefaSerializer(data=request.data)

            if serializer.is_valid():
                serializer.save()
                logger.info(f"Tarefa criada: {serializer.data['id']}")
                return Response(
                    serializer.data,
                    status=status.HTTP_201_CREATED
                )

            logger.warning(f"Validação falhou: {serializer.errors}")
            return Response(
                serializer.errors,
                status=status.HTTP_400_BAD_REQUEST
            )
        except Exception as e:
            logger.error(f"Erro ao criar tarefa: {str(e)}")
            return Response(
                {'error': 'Erro interno do servidor.'},
                status=status.HTTP_500_INTERNAL_SERVER_ERROR
            )
```

# Capítulo 6: Testes Práticos Completos

## 6.1. Roteiro de Testes no Postman/Insomnia

### Teste 1: Criação Bem-Sucedida

Método: POST  
URL: http://localhost:8000/api/tarefas/  
Headers:  
Content-Type: application/json  
Body (raw JSON):  
{  
 "titulo": "Implementar método POST",  
 "concluida": false  
}

 Resultado Esperado:  
Status: 201 Created  
Body:  
{  
 "id": 1,  
 "titulo": "Implementar método POST",  
 "concluida": false,  
 "criada\_em": "2025-11-29T15:30:00Z"  
}

## Teste 2: Validação - Título Vazio

Body:

```
{  
  "titulo": "",  
  "concluida": false  
}
```

✗ Resultado Esperado:

Status: 400 Bad Request

Body:

```
{  
  "titulo": [  
    "Este campo não pode ser em branco."  
  ]  
}
```

## Teste 3: Validação - Campo Obrigatório

Body:

```
{  
  "concluida": true  
}
```

✗ Resultado Esperado:

Status: 400 Bad Request

Body:

```
{  
  "titulo": [  
    "Este campo é obrigatório."  
  ]  
}
```

## Teste 4: Validação - Tipo Incorreto

Body:

```
{  
  "titulo": "Minha tarefa",  
  "concluida": "sim"  
}
```

✗ Resultado Esperado:

Status: 400 Bad Request

Body:

```
{  
  "concluida": [  
    "Insira um valor booleano válido."  
  ]  
}
```

## Teste 5: Validação - max\_length

Body:

```
{  
  "titulo": "a" (repita 300 vezes),  
  "concluida": false  
}
```

✗ Resultado Esperado:

Status: 400 Bad Request

```
Body:  
{  
  "titulo": [  
    "Certifique-se de que este campo não tenha mais de 200 caracteres."  
  ]  
}
```

### Teste 6: Criar e Listar

1. Criar 3 tarefas (POST)
2. Listar todas (GET /api/tarefas/)

 Resultado: A lista deve conter as 3 tarefas criadas

## 6.2. Teste com cURL

```
# Criar tarefa  
curl -X POST http://localhost:8000/api/tarefas/ \  
-H "Content-Type: application/json" \  
-d '{"titulo": "Testar via cURL", "concluida": false}'  
  
# Listar tarefas  
curl http://localhost:8000/api/tarefas/
```

## 6.3. Teste com Python Requests

```
import requests  
  
# URL base  
base_url = "http://localhost:8000/api/tarefas/"  
  
# Teste 1: Criar tarefa  
response = requests.post(  
    base_url,  
    json={  
        "titulo": "Tarefa via Python",  
        "concluida": False  
    }  
)  
  
print(f"Status: {response.status_code}")  
print(f"Resposta: {response.json()}")  
  
# Teste 2: Listar tarefas  
response = requests.get(base_url)  
tarefas = response.json()  
print(f"Total de tarefas: {len(tarefas)}")
```

# Capítulo 7: Padrões e Boas Práticas

## 7.1. Estrutura de Código

```
# ✓ BOM: Métodos organizados e documentados
class ListaTarefasAPIView(APIView):
    """
        Gerencia coleção de tarefas.

        GET: Lista tarefas
        POST: Cria tarefa
    """

    def get(self, request, format=None):
        """Lista todas as tarefas."""
        tarefas = Tarefa.objects.all()
        serializer = TarefaSerializer(tarefas, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        """Cria nova tarefa."""
        serializer = TarefaSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
        return Response(
            serializer.data,
            status=status.HTTP_201_CREATED
        )
    return Response(
        serializer.errors,
        status=status.HTTP_400_BAD_REQUEST
    )
```

## 7.2. DRY (Don't Repeat Yourself)

```
# ✗ RUIM: Repetição de código
def post(self, request):
    if 'titulo' not in request.data:
        return Response({'error': 'titulo required'})
    if len(request.data['titulo']) > 200:
        return Response({'error': 'titulo too long'})
    # ...

# ✓ BOM: Usar Serializer para validação
def post(self, request):
    serializer = TarefaSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
    return Response(serializer.data, status=201)
    return Response(serializer.errors, status=400)
```

## 7.3. Consistência de Respostas

```
# ✓ Sempre use status codes do DRF
from rest_framework import status

return Response(data, status=status.HTTP_201_CREATED)

# ✗ Evite números mágicos
return Response(data, status=201)
```

## 7.4. Tratamento de Erros Gracioso

```
# ✅ BOM: Sempre trate erros
try:
    serializer.save()
except Exception as e:
    logger.error(f"Erro: {e}")
    return Response(
        {'error': 'Erro ao processar requisição'},
        status=500
    )

# ❌ RUIM: Deixar exceção vazar
serializer.save() # Pode quebrar sem aviso
```



## Exercícios Práticos

### Exercício 1: Campo de Prioridade

Adicione um campo `prioridade` ao Model:

```
PRIORIDADE_CHOICES = [
    ('baixa', 'Baixa'),
    ('media', 'Média'),
    ('alta', 'Alta'),
]

prioridade = models.CharField(
    max_length=10,
    choices=PRIORIDADE_CHOICES,
    default='media'
)
```

1. Crie a migração
2. Adicione ao Serializer
3. Teste criando tarefas com prioridades diferentes
4. Adicione validação para aceitar apenas valores válidos

### Exercício 2: Validação de Data

Adicione um campo `prazo`:

```
prazo = models.DateField(null=True, blank=True)
```

Implemente validação customizada:

- O prazo não pode ser no passado
- Se `concluida=True`, prazo não é obrigatório
- Se `concluida=False`, prazo é obrigatório

## Exercício 3: API de Estatísticas

Crie um endpoint /api/tarefas/estatisticas/ que retorne:

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

Dicas:

- Use APIView com método get
- Use agregação: Tarefa.objects.aggregate()
- Calcule a taxa de conclusão

## Exercício 4: Soft Delete

Implemente exclusão suave:

- Adicione campo deletada = models.BooleanField(default=False)
- Modifique o GET para não mostrar tarefas deletadas
- Crie endpoint PATCH para marcar como deletada

## Resumo da Apostila

### O que aprendemos:

1.  Implementar método POST para criação de recursos
2.  Validação automática com Serializers
3.  Validações customizadas (validate\_<campo> e validate())
4.  Tratamento de erros e status codes apropriados
5.  Testes completos de criação e validação
6.  Boas práticas de código e padrões REST

### Fluxo Completo Implementado:

GET /api/tarefas/ → Lista todas as tarefas  
POST /api/tarefas/ → Cria nova tarefa (com validação)

### Próximos passos (Apostila 3):

- Operações em recursos individuais (URLs com pk)
- PUT (atualização total)
- PATCH (atualização parcial)
- DELETE (exclusão)
- Tratamento de 404 Not Found



## Referências

- [Django REST Framework Serializers](#)
- [Django REST Framework Validation](#)
- [HTTP Status Codes](#)
- [REST API Design Best Practices](#)



## Checklist de Verificação

Antes de prosseguir para a Apostila 3, certifique-se de que:

- [ ] O método POST está funcionando corretamente
- [ ] Validações estão retornando erros apropriados
- [ ] Status codes corretos (201 para sucesso, 400 para erro)
- [ ] Consegue criar tarefas via Postman/Insomnia
- [ ] Consegue listar tarefas criadas via POST
- [ ] Entende o fluxo: dados → validação → salvamento
- [ ] Testou todos os cenários de erro



**Parabéns! Você implementou criação de recursos com validação profissional!**

Na próxima apostila, vamos trabalhar com recursos individuais e implementar atualização e exclusão.