Apostila 3: Manipulação de Recursos Individuais (PUT, PATCH, DELETE)

🎯 Objetivos de Aprendizagem

Ao final desta apostila, você será capaz de:

- **Diferenciar** operações em coleções vs. recursos individuais
- Trabalhar com **URLs dinâmicas** (pk na URL)
- Implementar **atualização total (PUT)** e **parcial (PATCH)**
- Implementar **exclusão de recursos (DELETE)**
- Tratar **erro 404 Not Found** adequadamente
- Compreender **idempotência** em APIs REST
- Criar um **CRUD completo** e funcional

Capítulo 1: Arquitetura REST - Coleção vs. Recurso Individual 🧩

1.1. O Paradigma de URLs RESTful

Em APIs REST, a URL define o escopo da operação:

ENDPOINTS DA API DE TAREFAS
📦 **COLEÇÃO:** /api/tarefas/
↳ Opera em **MÚLTIPLOS** recursos
GET: Lista todas as tarefas
POST: Cria nova tarefa
📄 **RECURSO INDIVIDUAL:** /api/tarefas/<pk>/
↳ Opera em **UM ÚNICO** recurso
GET: Busca tarefa específica
PUT: Atualiza tarefa integralmente
PATCH: Atualiza tarefa parcialmente
DELETE: Remove tarefa

1.2. Comparação Detalhada

Aspecto	Coleção	Recurso Individual
URL	/api/tarefas/	/api/tarefas/5/
Identificador	Nenhum	**pk (Primary Key)** na URL
GET	Lista todos	Busca um específico
POST	Cria novo	❌ Não usado
PUT	❌ Não usado	Atualiza completamente
PATCH	❌ Não usado	Atualiza parcialmente
DELETE	❌ Não usado	Remove o recurso

1.3. Exemplo Prático de Uso

```
# COLEÇÃO: Criar nova tarefa
POST /api/tarefas/
{"titulo": "Estudar REST", "concluida": false}
→ 201 Created
{"id": 42, "titulo": "Estudar REST", ...}

# RECURSO: Buscar tarefa criada
GET /api/tarefas/42/
→ 200 OK
{"id": 42, "titulo": "Estudar REST", ...}

# RECURSO: Atualizar parcialmente
PATCH /api/tarefas/42/
{"concluida": true}
→ 200 OK
{"id": 42, "titulo": "Estudar REST", "concluida": true, ...}
```

```
# RECURSO: Deletar
DELETE /api/tarefas/42/
→ 204 No Content
```

1.4. O Conceito de Atomicidade

Operações em recursos individuais são **atômicas**, focadas em uma única entidade, garantindo que a ação seja aplicada exatamente onde é destinada.

```
# ✅ ATÔMICO: Afeta apenas a tarefa 5
PATCH /api/tarefas/5/
{"concluida": true}

# ❌ NÃO ATÔMICO: Afeta múltiplas tarefas
# PATCH /api/tarefas/{"concluida": true}  # Isso seria uma operação em lote (bulk), não o padrão para coleções REST.
```

Capítulo 2: URLs Dinâmicas e Captura de Parâmetros 🔗

2.1. Path Parameters (pk na URL)

Django permite capturar valores da URL usando o recurso de **Path Converters**:

```
# core/urls.py
from django.urls import path
from .views import ListaTarefasAPIView, DetalheTarefaAPIView

app_name = 'core'
urlpatterns = [
    # COLEÇÃO: /api/tarefas/
    path('tarefas/',
         ListaTarefasAPIView.as_view(),
         name='lista-tarefas'),

    # RECURSO INDIVIDUAL: /api/tarefas/<pk>/
    path('tarefas/<int:pk>/',
         DetalheTarefaAPIView.as_view(),
         name='detalhe-tarefa'),
    #          ^^^^^^^^
    #          Captura o ID e passa para a view
]
```

Explicação:

- **<int:pk>**: Captura um número inteiro da URL.
- **pk**: Nome do parâmetro (Primary Key) que será passado para o método da View.
- **int**: Tipo de conversão (Path Converter), garante que o valor seja um número.

Exemplos de Matching:

- ✅ /api/tarefas/1/ $\rightarrow pk = 1$
- ✅ /api/tarefas/42/ $\rightarrow pk = 42$
- ❌ /api/tarefas/abc/ $\rightarrow$ Não match (não é int)

2.2. Outros Tipos de Path Parameters

- path('categorias/<str:nome>/', ...): Captura uma string.
- path('produtos/<uuid:produto_id>/', ...): Captura um UUID.

- path('posts/<slug:slug>/', ...): Captura um slug (texto separado por hífens/underscores).

2.3. Recebendo <pk> na View

O valor capturado é passado como argumento posicional (ou nomeado) para o método HTTP da View.

```
class DetalheTarefaAPIView(APIView):
    def get(self, request, pk, format=None):
        #               ^^
        #               Parâmetro capturado da URL
        print(f"Buscando tarefa com ID: {pk}")
        # ...
```

Capítulo 3: Busca Segura e Tratamento de 404 🔍

3.1. O Problema de IDs Inexistentes

Se você usar Tarefa.objects.get(pk=999) e o recurso não existir, o Django lança uma exceção DoesNotExist, que, se não for tratada, resulta em um **erro 500 (Internal Server Error)**. Para uma API REST, o esperado é um **404 Not Found**.

- ✅ **SOLUÇÃO 1: Try/Except manual**
  Python
  ```
  try:
      tarefa = Tarefa.objects.get(pk=pk)
  except Tarefa.DoesNotExist:
      return Response({'error': 'Tarefa não encontrada'}, status=404)
  ```

- ✅ **SOLUÇÃO 2: get_object_or_404 (MELHOR)**
  Python
  ```
  from django.shortcuts import get_object_or_404
  tarefa = get_object_or_404(Tarefa, pk=pk)
  # Automaticamente lança uma exceção Http404 se não encontrar
  ```

3.2. Método Auxiliar get_object (DRY)

Para evitar repetir a lógica de busca e tratamento de 404 em todos os métodos (GET, PUT, PATCH, DELETE), usamos um método auxiliar na View.

```python
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
# ...

class DetalheTarefaAPIView(APIView):
    """
    View para operações em recurso individual.
    """
    def get_object(self, pk):
        """
        Busca a tarefa pelo ID e retorna 404 se não encontrada.
        """
        return get_object_or_404(Tarefa, pk=pk)

    # ... Métodos GET, PUT, PATCH, DELETE usarão self.get_object(pk)
```

Vantagens:

- ✅ **DRY** (Don't Repeat Yourself)
- ✅ **Tratamento automático de 404**
- ✅ **Consistência** na busca em todos os métodos

Capítulo 4: GET de Detalhe (Buscar Um Recurso) 📄

4.1. Implementação do GET Individual

```
class DetalheTarefaAPIView(APIView):
    def get_object(self, pk):
        """Busca tarefa ou retorna 404."""
        return get_object_or_404(Tarefa, pk=pk)

    def get(self, request, pk, format=None):
        """
        Retorna os dados de uma tarefa específica.

        Args:
            pk: ID da tarefa na URL
        Returns:
            200 OK: Tarefa encontrada
            404 Not Found: Tarefa não existe
        """
        # 1. BUSCAR: Usa método auxiliar (trata 404)
        tarefa = self.get_object(pk)

        # 2. SERIALIZAR: Converte objeto único (sem many=True)
        serializer = TarefaSerializer(tarefa)

        # 3. RESPONDER: Retorna JSON com status 200
        return Response(serializer.data, status=status.HTTP_200_OK)
```

4.2. Diferença: GET Coleção vs. GET Individual

GET Coleção (Lista)	**GET Individual (Um)**
tarefas = Tarefa.objects.all() (**QuerySet**)	tarefa = self.get_object(pk) (**Instância única**)
serializer = TarefaSerializer(tarefas, many=True)	serializer = TarefaSerializer(tarefa) (**Sem** many=True)

Capítulo 5: PUT - Atualização Total 🔄

5.1. Conceito de PUT

PUT significa **Substituição Completa** do Recurso. A carga enviada pelo cliente deve conter **TODOS** os campos editáveis.

Exemplo:

ANTES:
{ "id": 5, "titulo": "Estudar Django", "concluida": false }

PUT /api/tarefas/5/
{ "titulo": "Estudar DRF", "concluida": true } # Campos faltantes seriam apagados/resetados

DEPOIS:
{ "id": 5, "titulo": "Estudar DRF", "concluida": true }

5.2. Características do PUT

Característica	Descrição
Idempotente	Fazer a mesma requisição N vezes resulta no mesmo estado final.
Completo	Deve enviar **TODOS** os campos editáveis.
Sobrescreve	Substitui o estado inteiro do recurso.
Status	200 OK (com corpo) ou 204 No Content (sem corpo).

5.3. Implementação do PUT

```
class DetalheTarefaAPIView(APIView):
    # ... def get_object(self, pk): ...

    def put(self, request, pk, format=None):
        """
        Atualiza tarefa completamente (substituição total).

        Exige que TODOS os campos editáveis sejam enviados.
        """
        # 1. BUSCAR: Obter o objeto existente
        tarefa = self.get_object(pk)

        # 2. SERIALIZAR: Passar objeto antigo E novos dados
        serializer = TarefaSerializer(tarefa, data=request.data)
        #                             ^^^^^  ^^^^^^^^^^^^^^^^
        #                             |      Nova versão
        #                             Versão atual

        # 3. VALIDAR: Checar se JSON está completo e válido
        if serializer.is_valid():
            # 4. SALVAR: Atualizar no banco
            serializer.save()

            # 5. RESPONDER: Retornar objeto atualizado
            return Response(serializer.data, status=status.HTTP_200_OK)

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

5.4. Fluxo de Atualização PUT

1. **Requisição:** PUT /api/tarefas/5/ {"titulo": "Novo título", ...}
2. **Busca:** self.get_object(5) $\rightarrow$ Obtém a instância atual da tarefa.
3. **Serialização:** TarefaSerializer(tarefa, data={...}) $\rightarrow$ Instancia o Serializer, informando o objeto **existente** e os **novos dados**.
4. **Validação:** is_valid() $\rightarrow$ Verifica se os dados são válidos **e completos**.
5. **Salvar:** serializer.save() $\rightarrow$ Executa a atualização no banco de dados.

Capítulo 6: PATCH - Atualização Parcial 🩹

6.1. Conceito de PATCH

PATCH significa **Atualização Parcial** do Recurso. A carga enviada pelo cliente deve conter **APENAS** os campos que deseja modificar.

Exemplo:

```
ANTES:
{ "id": 5, "titulo": "Estudar DRF", "concluida": false }

PATCH /api/tarefas/5/
{ "concluida": true } # Apenas este campo é alterado

DEPOIS:
{ "id": 5, "titulo": "Estudar DRF", "concluida": true } # O título é mantido
```

6.2. Características do PATCH

Característica	Descrição
Idempotente	Sim, se a operação for bem definida (ex: marcar como concluída N vezes resulta em concluída).
Parcial	Deve enviar **SOMENTE** os campos que serão alterados.
Merge	Mescla (combina) os novos dados com o estado atual do recurso.
Status	200 OK.

6.3. Implementação do PATCH

Para o PATCH funcionar corretamente, é necessário passar o argumento partial=True na instanciação do Serializer, permitindo que campos obrigatórios sejam omitidos na requisição.

```
class DetalheTarefaAPIView(APIView):
    # ... def get_object(self, pk): ...

    def patch(self, request, pk, format=None):
        """
        Atualiza tarefa parcialmente (merge).

        Permite enviar apenas os campos que serão modificados.
        """
        # 1. BUSCAR: Obter o objeto existente
        tarefa = self.get_object(pk)

        # 2. SERIALIZAR: Passar objeto, novos dados E partial=True
        serializer = TarefaSerializer(
            tarefa,
            data=request.data,
            partial=True # <--- ESSENCIAL PARA O PATCH
        )

        # 3. VALIDAR
        if serializer.is_valid():
            # 4. SALVAR (aplica apenas os campos recebidos)
            serializer.save()

            # 5. RESPONDER
            return Response(serializer.data, status=status.HTTP_200_OK)

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

Capítulo 7: DELETE - Remoção de Recurso 🗑️

7.1. Conceito de DELETE

DELETE é usado para remover um recurso específico identificado pela URL. É uma das operações mais diretas.

7.2. Características do DELETE

Característica	Descrição
Idempotente	Sim. Tentar deletar um recurso que já foi deletado resulta no mesmo estado (recurso não existe).
Status	**204 No Content** é o padrão. Indica sucesso na operação, mas sem corpo de resposta.

7.3. Implementação do DELETE

```
class DetalheTarefaAPIView(APIView):
    # ... def get_object(self, pk): ...

    def delete(self, request, pk, format=None):
        """
        Remove um recurso específico.
        """
        # 1. BUSCAR: Obter o objeto (trata 404 se não existir)
        tarefa = self.get_object(pk)

        # 2. DELETAR
        tarefa.delete()

        # 3. RESPONDER: 204 No Content (sucesso sem corpo de resposta)
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Capítulo 8: DRF: Implementação Completa e URLs

(Os capítulos 8 e 9 originais foram mesclados e renumerados para manter o fluxo)

8.1. Arquivo de Views Completo

```
# core/views.py
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
# ... (Importar Model e Serializer)

class DetalheTarefaAPIView(APIView):
    """
    View para operações em recurso individual.
    GET, PUT, PATCH, DELETE /api/tarefas/<pk>/
    """
    def get_object(self, pk):
        return get_object_or_404(Tarefa, pk=pk)

    # 4. GET (Buscar)
    def get(self, request, pk, format=None):
        tarefa = self.get_object(pk)
        serializer = TarefaSerializer(tarefa)
        return Response(serializer.data, status=status.HTTP_200_OK)

    # 5. PUT (Atualização Total)
    def put(self, request, pk, format=None):
        tarefa = self.get_object(pk)
        serializer = TarefaSerializer(tarefa, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # 6. PATCH (Atualização Parcial)
    def patch(self, request, pk, format=None):
        tarefa = self.get_object(pk)
        serializer = TarefaSerializer(
            tarefa,
            data=request.data,
```

```
            partial=True  # Permite omissão de campos
        )
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # 7. DELETE (Remoção)
    def delete(self, request, pk, format=None):
        tarefa = self.get_object(pk)
        tarefa.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

8.2. URLs Completas

```
# core/urls.py
from django.urls import path
from .views import ListaTarefasAPIView, DetalheTarefaAPIView

app_name = 'core'
urlpatterns = [
    # Coleção: /api/tarefas/ (POST, GET - Lista)
    path('tarefas/',
         ListaTarefasAPIView.as_view(),
         name='lista-tarefas'),

    # Recurso Individual: /api/tarefas/<pk>/ (GET, PUT, PATCH, DELETE)
    path('tarefas/<int:pk>/',
         DetalheTarefaAPIView.as_view(),
         name='detalhe-tarefa'),
]
```

Capítulo 9: Roteiro de Testes Completo 🧪

(O conteúdo deste capítulo permanece inalterado por estar perfeito para o propósito.)

9.1. Preparação: Criar Tarefa de Teste

POST /api/tarefas/ $\rightarrow$ 201 Created$ (Anote o ID, ex: 10)

9.2. Teste 1: GET Individual (Buscar Tarefa)

GET /api/tarefas/10/ $\rightarrow$ 200$ OK

9.3. Teste 2: GET com ID Inexistente (404)

GET /api/tarefas/999/ $\rightarrow$ 404$ Not Found

9.4. Teste 3: PATCH Parcial (Sucesso)

PATCH /api/tarefas/10/ {"concluida": true} $\rightarrow$ 200$ OK

9.5. Teste 4: PUT Incompleto (Falha Esperada)

PUT /api/tarefas/10/ {"concluida": false} $\rightarrow$ 400$ Bad Request (Falta o campo titulo)

9.6. Teste 5: PUT Completo (Sucesso)

PUT /api/tarefas/10/ {"titulo": "Tarefa atualizada via PUT", "concluida": false} $\rightarrow$ 200$ OK

9.7. Teste 6: DELETE (Remoção)

DELETE /api/tarefas/10/ $\rightarrow$ 204$ No Content

9.8. Teste 7: Confirmação de Exclusão (404)

GET /api/tarefas/10/ $\rightarrow$ 404$ Not Found

9.9. Tabela Resumo dos Testes

Teste	Método	URL	Body	Status Esperado	Validação
1	POST	/api/tarefas/	{"titulo": "..."}	201	Criar tarefa
2	GET	/api/tarefas/10/	-	200	Buscar criada
3	GET	/api/tarefas/999/	-	404	ID inexistente
4	PATCH	/api/tarefas/10/	{"concluida": true}	200	Atualização parcial
5	PUT	/api/tarefas/10/	{"concluida": false}	400	PUT incompleto
6	PUT	/api/tarefas/10/	{"titulo": "...", "concluida": false}	200	PUT completo
7	DELETE	/api/tarefas/10/	-	204	Remover
8	GET	/api/tarefas/10/	-	404	Confirma exclusão

Capítulo 10: Conceitos Avançados 🚀

10.1. Idempotência em APIs REST

Método	Idempotente?	Explicação
GET	✅ Sim	Ler N vezes não muda o estado.
POST	❌ Não	Criar N vezes $\rightarrow N$ recursos.
PUT	✅ Sim	Substituir N vezes $\rightarrow$ mesmo estado final.
PATCH	✅ Sim	Modificar N vezes $\rightarrow$ mesmo estado (quando bem implementado).
DELETE	✅ Sim	Deletar N vezes $\rightarrow$ recurso não existe.

10.2. Safe Methods (Métodos Seguros)

Definição: Métodos que não modificam o estado do servidor. **Apenas GET e HEAD são considerados seguros.**

- **Implicações:** Métodos seguros podem ser **cacheados** e **pré-carregados**.

10.3. Validação Específica por Método

(O trecho de código e explicação sobre como usar self.context.get('request') para aplicar validações condicionais baseadas no método HTTP (POST, PATCH, etc.) está correto e é um ótimo recurso avançado.)

10.4. ETags e Concorrência Otimista

(O trecho de código e explicação sobre como usar um campo de **versão** e retornar 409 Conflict para gerenciar atualizações simultâneas está correto e é um excelente conteúdo avançado.)

📝 Exercícios Práticos

(Conteúdo mantido - Ótimos exercícios para fixação.)

- Exercício 1: Adicionar Campo de Data de Conclusão (Lógica de preenchimento/limpeza no Serializer)
- Exercício 2: Endpoint de Duplicação (POST /api/tarefas/<pk>/duplicar/)
- Exercício 3: PATCH em Lote (PATCH /api/tarefas/concluir-todas/)
- Exercício 4: Validação de Estado (Regra: tarefas "alta" prioridade só podem ser concluídas via PUT)

🎓 Resumo da Apostila

O que aprendemos:

- ✅ Diferença entre operações em coleção vs. recurso individual
- ✅ URLs dinâmicas com captura de pk
- ✅ Tratamento de 404 com get_object_or_404
- ✅ GET individual (buscar um recurso)
- ✅ **PUT** (atualização **total** com campos completos)
- ✅ **PATCH** (atualização **parcial** com partial=True)
- ✅ **DELETE** (remoção com status 204)
- ✅ Conceitos de **idempotência** e **métodos seguros**

CRUD Completo Implementado:

- **C** - CREATE: POST /api/tarefas/
- **R** - READ: GET /api/tarefas/ (lista) $\mid$ GET /api/tarefas/<pk>/ (individual)
- **U** - UPDATE: PUT /api/tarefas/<pk>/ (total) $\mid$ PATCH /api/tarefas/<pk>/ (parcial)
- **D** - DELETE: DELETE /api/tarefas/<pk>/

Próximos passos (Apostila 4):

- Autenticação com JWT
- Proteção de rotas com IsAuthenticated
- Injeção automática do usuário logado

🔍 Checklist de Verificação

Antes de prosseguir para a Apostila 4:

- [ ] GET individual retorna tarefa específica
- [ ] GET com ID inexistente retorna 404
- [ ] PATCH altera apenas campos enviados
- [ ] PUT exige todos os campos obrigatórios
- [ ] PUT com campos incompletos retorna 400
- [ ] DELETE remove tarefa e retorna 204
- [ ] GET após DELETE retorna 404
- [ ] Entende diferença entre PUT e PATCH
- [ ] Entende conceito de idempotência

🎉 **Parabéns! Você implementou um CRUD completo e funcional!**