

6.1 Adição de Endpoints POST, PUT e DELETE

No Capítulo 5, criamos uma API REST simples com endpoints GET para listar e recuperar tarefas, utilizando Node.js com Express. Agora, expandiremos essa API para suportar operações de criação (POST), atualização (PUT e PATCH) e exclusão (DELETE), completando o conjunto CRUD (Create, Read, Update, Delete) essencial para APIs RESTful. Esses endpoints permitirão manipular recursos de forma dinâmica, seguindo os princípios REST introduzidos no Capítulo 3 (interface uniforme, sem estado) e as boas práticas de modelagem de dados do Capítulo 4 (JSON, HATEOAS, validação).

Por que POST, PUT e DELETE?

- **POST:** Cria novos recursos, retornando o recurso criado com um código de status 201 Created.
- **PUT:** Atualiza um recurso existente completamente, substituindo todos os seus campos.
- **PATCH:** Atualiza parcialmente um recurso, modificando apenas os campos fornecidos.
- **DELETE:** Remove um recurso, retornando 204 No Content em caso de sucesso.

Esses métodos HTTP são fundamentais para manipular recursos de forma RESTful, garantindo que a API seja previsível e escalável. Para manter a simplicidade, continuaremos usando um array em memória como “banco de dados”, mas discutiremos como integrar um banco real (ex.: PostgreSQL) no futuro.

Estrutura do Recurso

O recurso “tarefa” terá a seguinte estrutura, conforme definido no Capítulo 5:

```
{
  "id": 1,
  "titulo": "Estudar REST APIs",
  "concluida": false,
  "prioridade": "alta",
  "descricao": "Revisar capítulos 3 a 5",
  "criadoEm": "2025-05-26T00:17:00Z",
  "atualizadoEm": null,
```

```
"tags": ["estudo", "programação"]
}
```

- **Campos:**

- `id`: Identificador único (gerado pelo servidor).
- `titulo`: Nome da tarefa (obrigatório, mínimo 3 caracteres).
- `concluida`: Estado da tarefa (booleano, padrão: `false`).
- `prioridade`: Nível de prioridade (alta, média, baixa).
- `descricao`: Detalhes adicionais (opcional, até 200 caracteres).
- `criadoEm`: Data de criação (ISO 8601).
- `atualizadoEm`: Data de última atualização (ISO 8601, opcional).
- `tags`: Lista de palavras-chave (opcional).

Implementação dos Endpoints

Vamos atualizar o código do Capítulo 5 para incluir os novos endpoints. Primeiro, instalaremos o **Joi**, uma biblioteca open-source para validação de dados:

```
npm install joi
```

Aqui está o código atualizado (`src/index.js`):

```
const express = require('express');
const Joi = require('joi');
const app = express();
const PORT = 3000;

app.use(express.json());

// Dados fictícios
let tarefas = [
  {
    id: 1,
    titulo: "Estudar REST APIs",
    concluida: false,
    prioridade: "alta",
    descricao: "Revisar capítulos 3 a 5",
    criadoEm: "2025-05-26T00:17:00Z",
    atualizadoEm: null,
```

```
    tags: ["estudo", "programação"]
  },
  {
    id: 2,
    titulo: "Fazer compras",
    concluida: true,
    prioridade: "média",
    descricao: "Comprar mantimentos para a semana",
    criadoEm: "2025-05-25T12:00:00Z",
    atualizadoEm: null,
    tags: ["compras"]
  }
];
```

```
// Função para gerar ID único
```

```
let ultimoId = 2;
```

```
const gerarId = () => ++ultimoId;
```

```
// Schema de validação com Joi
```

```
const tarefaSchema = Joi.object({
  titulo: Joi.string().min(3).max(100).required(),
  concluida: Joi.boolean().default(false),
  prioridade: Joi.string().valid('alta', 'média', 'baixa').default('média'),
  descricao: Joi.string().max(200).allow('').optional(),
  tags: Joi.array().items(Joi.string().max(20)).optional()
});
```

```
// Middleware para validação
```

```
const validarTarefa = (req, res, next) => {
  const { error } = tarefaSchema.validate(req.body, { abortEarly: false });
  if (error) {
    const erros = error.details.map(err => err.message);
    return res.status(400).json({ erros });
  }
  next();
};
```

```
// GET /tarefas
```

```

app.get('/tarefas', (req, res) => {
  const tarefasComLinks = tarefas.map(tarefa => ({
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      pendentes: `/tarefas/pendentes`
    }
  }));
  res.json({
    tarefas: tarefasComLinks,
    meta: {
      total: tarefas.length,
      page: 1,
      limit: 10
    }
  });
});

// GET /tarefas/:id
app.get('/tarefas/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const tarefa = tarefas.find(t => t.id === id);
  if (!tarefa) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  res.json({
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      all: `/tarefas`,
      pendentes: `/tarefas/pendentes`
    }
  });
});

```

```

// GET /tarefas/pendentes
app.get('/tarefas/pendentes', (req, res) => {
  const tarefasPendentes = tarefas.filter(t => !t.concluida).map(tarefa =>
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      all: `/tarefas`
    }
  ));
  res.json({
    tarefas: tarefasPendentes,
    meta: {
      total: tarefasPendentes.length,
      page: 1,
      limit: 10
    }
  });
});

// POST /tarefas
app.post('/tarefas', validarTarefa, (req, res) => {
  const tarefa = {
    id: gerarId(),
    ...req.body,
    criadoEm: new Date().toISOString(),
    atualizadoEm: null
  };
  tarefas.push(tarefa);
  res.status(201).json({
    ...tarefa,
    _links: {
      self: `/tarefas/${tarefa.id}`,
      all: `/tarefas`
    }
  });
});

// PUT /tarefas/:id

```

```
app.put('/tarefas/:id', validarTarefa, (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const index = tarefas.findIndex(t => t.id === id);
  if (index === -1) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  const tarefaAtualizada = {
    id,
    ...req.body,
    criadoEm: tarefas[index].criadoEm,
    atualizadoEm: new Date().toISOString()
  };
  tarefas[index] = tarefaAtualizada;
  res.json({
    ...tarefaAtualizada,
    _links: {
      self: `/tarefas/${tarefaAtualizada.id}`,
      all: `/tarefas`
    }
  });
});
```

```
// PATCH /tarefas/:id
app.patch('/tarefas/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const index = tarefas.findIndex(t => t.id === id);
  if (index === -1) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  const tarefaAtual = tarefas[index];
  const dadosAtualizados = {
    ...tarefaAtual,
```

```

    ...req.body,
    atualizadoEm: new Date().toISOString()
  };
  const { error } = tarefaSchema.validate(dadosAtualizados, { abortEarly:
  if (error) {
    const erros = error.details.map(err => err.message);
    return res.status(400).json({ erros });
  }
  tarefas[index] = dadosAtualizados;
  res.json({
    ...dadosAtualizados,
    _links: {
      self: `/tarefas/${dadosAtualizados.id}`,
      all: `/tarefas`
    }
  });
});

// DELETE /tarefas/:id
app.delete('/tarefas/:id', (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const index = tarefas.findIndex(t => t.id === id);
  if (index === -1) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  tarefas.splice(index, 1);
  res.status(204).send();
});

// Inicia o servidor
app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

```

Explicação do Código

1. **Dependências:**
2. `express`: Framework para a API.
3. `joi`: Validação de dados.
4. **Dados Fictícios:**
5. O array `tarefas` simula um banco de dados.
6. `gerarId()` cria IDs únicos incrementais.
7. **Validação com Joi:**
8. `tarefaSchema` define regras para os campos (ex.: `titulo` obrigatório, `prioridade` com valores fixos).
9. O middleware `validarTarefa` verifica requisições POST e PUT.
10. **Endpoints:**
11. **POST /tarefas**: Cria uma nova tarefa, gera um ID e adiciona `criadoEm`. Retorna 201 Created com o recurso criado.
12. **PUT /tarefas/:id**: Substitui a tarefa completamente, preservando `criadoEm` e atualizando `atualizadoEm`.
13. **PATCH /tarefas/:id**: Atualiza parcialmente, mesclando os dados fornecidos com os existentes e validando o resultado.
14. **DELETE /tarefas/:id**: Remove a tarefa, retornando 204 No Content.
15. **HATEOAS:**
16. Respostas incluem links (`self`, `all`, `pendentes`) para navegação.
17. **Códigos de Status:**
18. 201 Created (POST).
19. 200 OK (PUT, PATCH).
20. 204 No Content (DELETE).
21. 400 Bad Request (validação ou ID inválido).
22. 404 Not Found (tarefa não encontrada).

Testando os Endpoints

1. **Inicie o servidor**: `bash npm run dev` (Assumindo Nodemon configurado no Capítulo 5.)
2. **Testes com cURL:**
3. **POST**: `bash curl -X POST http://localhost:3000/tarefas \ -H "Content-Type: application/json" \ -d '{"titulo":"Correr 5km","prioridade":"baixa","descricao":"Treino matinal","tags":["exercício"]}'` **Saída**: `json { "id": 3,`


```
"titulo": "Correr 5km", "concluida": false, "prioridade":
"baixa", "descricao": "Treino matinal", "tags":
["exercício"], "criadoEm": "2025-05-26T03:37:00Z",
"atualizadoEm": null, "_links": { "self": "/tarefas/3",
"all": "/tarefas" } }
```

4. **PUT:** bash curl -X PUT http://localhost:3000/tarefas/1 \ -H
"Content-Type: application/json" \ -d '{"titulo":"Estudar
REST APIs Avançado", "concluida":true, "prioridade":"alta", "descricao":"Focar
em validação"}' **Saída:** json { "id": 1, "titulo": "Estudar
REST APIs Avançado", "concluida": true, "prioridade":
"alta", "descricao": "Focar em validação", "criadoEm":
"2025-05-26T00:17:00Z", "atualizadoEm":
"2025-05-26T03:38:00Z", "_links": { "self": "/tarefas/1",
"all": "/tarefas" } }
5. **PATCH:** bash curl -X PATCH http://localhost:3000/tarefas/2 \
-H "Content-Type: application/json" \ -d
'{"concluida":false}' **Saída:** json { "id": 2, "titulo": "Fazer
compras", "concluida": false, "prioridade": "média",
"descricao": "Comprar mantimentos para a semana",
"criadoEm": "2025-05-25T12:00:00Z", "atualizadoEm":
"2025-05-26T03:39:00Z", "tags": ["compras"], "_links": {
"self": "/tarefas/2", "all": "/tarefas" } }

DELETE: bash curl -X DELETE http://localhost:3000/tarefas/2
Saída: Nenhuma (status 204).

Teste com REST Client (VS Code): Crie um arquivo test.http: `` POST
http://localhost:3000/tarefas HTTP/1.1 Content-Type: application/json

```
{ "titulo": "Correr 5km", "prioridade": "baixa", "descricao": "Treino matinal", "tags":  
["exercício"] }
```

###

PUT http://localhost:3000/tarefas/1 HTTP/1.1 Content-Type: application/json

```
{ "titulo": "Estudar REST APIs Avançado", "concluida": true, "prioridade": "alta",  
"descricao": "Focar em validação" }
```

###

PATCH http://localhost:3000/tarefas/2 HTTP/1.1 Content-Type: application/json

```
{ "concluida": false }
```

###

DELETE http://localhost:3000/tarefas/2 HTTP/1.1 ``

Equivalência com Flask

Para quem prefere Python, aqui está o endpoint POST em Flask como exemplo:

```
from flask import Flask, jsonify, request
from datetime import datetime
```

```
app = Flask(__name__)
```

```
tarefas = [...] # Mesmo array do exemplo Express
ultimo_id = 2
```

```
@app.route('/tarefas', methods=['POST'])
```

```
def criar_tarefa():
```

```
    if not request.is_json:
```

```
        return jsonify({"erro": "Content-Type deve ser application/json"})
```

```
    dados = request.get_json()
```

```
    if not dados.get('titulo') or len(dados.get('titulo', '')) < 3:
```

```
        return jsonify({"erro": "Título deve ter pelo menos 3 caracteres"})
```

```
    tarefa = {
```

```
        "id": ultimo_id + 1,
```

```
        "titulo": dados['titulo'],
```

```
        "concluida": dados.get('concluida', False),
```

```
        "prioridade": dados.get('prioridade', 'média'),
```

```
        "descricao": dados.get('descricao', ''),
```

```
        "tags": dados.get('tags', []),
```

```
        "criadoEm": datetime.utcnow().isoformat() + 'Z',
```

```
        "atualizadoEm": None
```

```
    }
```

```
    global ultimo_id
```

```

    ultimo_id += 1
    tarefas.append(tarefa)
    return jsonify({
        **tarefa,
        "_links": {
            "self": f"/tarefas/{tarefa['id']}",
            "all": "/tarefas"
        }
    }), 201

if __name__ == '__main__':
    app.run(port=3000, debug=True)

```

6.2 Validação de Dados com Joi

Validação é crucial para garantir que os dados recebidos sejam consistentes e seguros. No código acima, usamos o **Joi**, uma biblioteca open-source para Node.js, que permite definir esquemas de validação declarativos.

Por que Validação?

- **Integridade:** Evita dados inválidos (ex.: título vazio, prioridade fora do permitido).
- **Segurança:** Reduz riscos de injeção de dados maliciosos.
- **Usabilidade:** Fornece mensagens de erro claras para os clientes.

Configuração do Joi

O schema `tarefaSchema` define as regras: - `titulo`: String, 3 a 100 caracteres, obrigatório. - `concluida`: Booleano, padrão `false`. - `prioridade`: String com valores `alta`, `média`, `baixa`, padrão `média`. - `descricao`: String até 200 caracteres, opcional. - `tags`: Array de strings até 20 caracteres, opcional.

Exemplo de erro de validação:

```

curl -X POST http://localhost:3000/tarefas \
  -H "Content-Type: application/json" \
  -d '{"titulo":""}'

```

Saída:

```
{
  "erros": [
    "\"titulo\" is not allowed to be empty",
    "\"titulo\" length must be at least 3 characters long"
  ]
}
```

Validação em PATCH

Para o endpoint PATCH, validamos o objeto resultante após mesclar os dados novos com os existentes, garantindo que o recurso atualizado ainda seja válido.

Alternativa para Flask

Em Python, a biblioteca **Pydantic** (open-source) oferece validação semelhante:

```
from pydantic import BaseModel, constr, ValidationError
from typing import List, Optional

class Tarefa(BaseModel):
    titulo: constr(min_length=3, max_length=100)
    concluida: bool = False
    prioridade: str = 'média'
    descricao: Optional[constr(max_length=200)] = ''
    tags: Optional[List[constr(max_length=20)]] = []

@app.route('/tarefas', methods=['POST'])
def criar_tarefa():
    try:
        tarefa = Tarefa(**request.get_json()).dict()
        tarefa['id'] = ultimo_id + 1
        tarefa['criadoEm'] = datetime.utcnow().isoformat() + 'Z'
        tarefa['atualizadoEm'] = None
        global ultimo_id
        ultimo_id += 1
        tarefas.append(tarefa)
        return jsonify({
```

```
        **tarefa,  
        "_links": {  
            "self": f"/tarefas/{tarefa['id']}",  
            "all": "/tarefas"  
        }  
    }), 201  
except ValidationError as e:  
    return jsonify({"erros": e.errors()}), 400
```

6.3 Tratamento de Erros e Respostas Consistentes

O tratamento de erros é essencial para criar APIs robustas e fáceis de usar. Boas práticas incluem: - **Códigos de Status**: Usar códigos HTTP apropriados (400, 404, 500, etc.). - **Mensagens Claras**: Fornecer detalhes sobre o erro sem expor informações sensíveis. - **Formato Consistente**: Todas as respostas de erro devem seguir o mesmo padrão.

Estrutura de Erros

Adotamos o seguinte formato para erros:

```
{  
  "erro": "Mensagem principal do erro",  
  "detalhes": ["Detalhe 1", "Detalhe 2"],  
  "campo": "Nome do campo problemático (opcional)"  
}
```

Ou, para erros de validação:

```
{  
  "erros": ["Mensagem de erro 1", "Mensagem de erro 2"]  
}
```

Middleware de Erros

Adicione um middleware global para capturar erros inesperados:

```
// Adicione antes de app.listen  
app.use((err, req, res, next) => {
```

```
console.error(err.stack);
res.status(500).json({
  erro: "Erro interno do servidor",
  detalhes: [err.message]
});
});
```

Exemplos de Erros

- **400 Bad Request** (validação): `json { "erros": ["\"titulo\" is required", "\"prioridade\" must be one of [alta, média, baixa]"] }`
- **404 Not Found**: `json { "erro": "Tarefa não encontrada" }`
- **500 Internal Server Error** (exemplo hipotético): `json { "erro": "Erro interno do servidor", "detalhes": ["Falha ao acessar o banco de dados"] }`

Boas Práticas

- **Evite Stack Traces**: Não exponha detalhes técnicos em produção.
- **Log de Erros**: Use `console.error` ou bibliotecas como Winston para registrar erros.
- **Documentação**: Inclua exemplos de erros na documentação da API.

6.4 Atividade Prática: Expansão da API de Tarefas

Nesta atividade, você expandirá a API de tarefas para incluir: 1. Um endpoint `POST /tarefas/batch` para criar múltiplas tarefas. 2. Um endpoint `PATCH /tarefas/:id/tags` para adicionar/remover tags. 3. Tratamento de erros para duplicatas (ex.: tarefa com título existente).

Requisitos

- Valide os dados com Joi.
- Retorne respostas com HATEOAS e códigos de status apropriados.
- Use o array `tarefas` como armazenamento.
- Teste com cURL ou REST Client.

Solução

Atualize `src/index.js` com os novos endpoints:

```
// Adicione ao schema existente
const tarefasBatchSchema = Joi.array().items(tarefaSchema).min(1).max(10)

// Middleware para validar batch
const validarTarefasBatch = (req, res, next) => {
  const { error } = tarefasBatchSchema.validate(req.body, { abortEarly: false })
  if (error) {
    const erros = error.details.map(err => err.message)
    return res.status(400).json({ erros })
  }
  next()
};

// POST /tarefas/batch
app.post('/tarefas/batch', validarTarefasBatch, (req, res) => {
  // Verifica duplicatas
  const titulosExistentes = tarefas.map(t => t.titulo.toLowerCase());
  const novosTitulos = req.body.map(t => t.titulo.toLowerCase());
  const duplicatas = novosTitulos.filter(t => titulosExistentes.includes(t))
  if (duplicatas.length > 0) {
    return res.status(409).json({
      erro: "Títulos duplicados encontrados",
      detalhes: duplicatas
    });
  }

  const novasTarefas = req.body.map(tarefa => ({
    id: gerarId(),
    ...tarefa,
    criadoEm: new Date().toISOString(),
    atualizadoEm: null
  }));
  tarefas.push(...novasTarefas);
  res.status(201).json({
    tarefas: novasTarefas.map(tarefa => ({
```

```

        ...tarefa,
        _links: {
            self: `/tarefas/${tarefa.id}`,
            all: `/tarefas`
        }
    })),
    meta: {
        total: novasTarefas.length
    }
});
});

// PATCH /tarefas/:id/tags
app.patch(`/tarefas/:id/tags`, (req, res) => {
    const id = parseInt(req.params.id);
    if (isNaN(id)) {
        return res.status(400).json({ erro: "ID deve ser um número" });
    }
    const index = tarefas.findIndex(t => t.id === id);
    if (index === -1) {
        return res.status(404).json({ erro: "Tarefa não encontrada" });
    }
    const schema = Joi.object({
        adicionar: Joi.array().items(Joi.string().max(20)).optional(),
        remover: Joi.array().items(Joi.string().max(20)).optional()
    }).or('adicionar', 'remover');
    const { error } = schema.validate(req.body);
    if (error) {
        return res.status(400).json({ erros: error.details.map(err => err.message) });
    }
    const tarefa = tarefas[index];
    let tags = tarefa.tags || [];
    if (req.body.adicionar) {
        tags = [...new Set([...tags, ...req.body.adicionar])]; // Evita duplicar
    }
    if (req.body.remover) {
        tags = tags.filter(tag => !req.body.remover.includes(tag));
    }
});

```



```

tarefa.tags = tags;
tarefa.atualizadoEm = new Date().toISOString();
tarefas[index] = tarefa;
res.json({
  ...tarefa,
  _links: {
    self: `/tarefas/${tarefa.id}`,
    all: `/tarefas`
  }
});
});

```

Testando

- POST /tarefas/batch:** bash curl -X POST
`http://localhost:3000/tarefas/batch \ -H "Content-Type: application/json" \ -d '[{"titulo":"Escrever relatório","prioridade":"alta","tags":["trabalho"]}, {"titulo":"Ler livro","prioridade":"baixa","tags":["lazer"]}]'`
Saída: json { "tarefas": [{ "id": 4, "titulo": "Escrever relatório", "concluida": false, "prioridade": "alta", "tags": ["trabalho"], "criadoEm": "2025-05-26T03:40:00Z", "atualizadoEm": null, "_links": { "self": "/tarefas/4", "all": "/tarefas" } }, { "id": 5, "titulo": "Ler livro", "concluida": false, "prioridade": "baixa", "tags": ["lazer"], "criadoEm": "2025-05-26T03:40:00Z", "atualizadoEm": null, "_links": { "self": "/tarefas/5", "all": "/tarefas" } }], "meta": { "total": 2 } }
- PATCH /tarefas/:id/tags:** bash curl -X PATCH
`http://localhost:3000/tarefas/1/tags \ -H "Content-Type: application/json" \ -d '{"adicionar":["api"],"remover":["estudo"]}'`
Saída: json { "id": 1, "titulo": "Estudar REST APIs", "concluida": false, "prioridade": "alta", "descricao": "Revisar capítulos 3 a 5", "criadoEm": "2025-05-26T00:17:00Z", "atualizadoEm": "2025-05-26T03:41:00Z", "tags": ["programação", "api"], "_links": { "self": "/tarefas/1", "all": "/tarefas" } }

3. **Erro de Duplicata:** bash curl -X POST
http://localhost:3000/tarefas/batch \ -H "Content-Type:
application/json" \ -d '["{"titulo":"Estudar REST
APIs","prioridade":"alta"}]'

Saída: json { "erro": "Títulos
duplicados encontrados", "detalhes": ["estudar rest apis"] }

Desafio Extra

Adicione um endpoint PUT /tarefas/:id/prioridade que atualiza apenas a prioridade, validando que o novo valor seja alta, média ou baixa.

Solução

```
app.put('/tarefas/:id/prioridade', (req, res) => {
  const id = parseInt(req.params.id);
  if (isNaN(id)) {
    return res.status(400).json({ erro: "ID deve ser um número" });
  }
  const index = tarefas.findIndex(t => t.id === id);
  if (index === -1) {
    return res.status(404).json({ erro: "Tarefa não encontrada" });
  }
  const schema = Joi.object({
    prioridade: Joi.string().valid('alta', 'média', 'baixa').required()
  });
  const { error } = schema.validate(req.body);
  if (error) {
    return res.status(400).json({ erros: error.details.map(err => err.message) });
  }
  tarefas[index].prioridade = req.body.prioridade;
  tarefas[index].atualizadoEm = new Date().toISOString();
  res.json({
    ...tarefas[index],
    _links: {
      self: `/tarefas/${tarefas[index].id}`,
      all: `/tarefas`
    }
  });
});
```

- **Teste:**

```
bash curl -X PUT http://localhost:3000/tarefas/1/prioridade \ -H "Content-Type: application/json" \ -d '{"prioridade": "média"}'
```

Conclusão

Este capítulo expandiu a API de tarefas com endpoints POST, PUT, PATCH e DELETE, implementando validação robusta com Joi e tratamento de erros consistente. A atividade prática consolidou esses conceitos, adicionando funcionalidades avançadas como criação em lote e manipulação de tags. Os princípios REST (Capítulo 3) e a modelagem de dados (Capítulo 4) foram aplicados, criando uma API funcional e escalável. Futuros capítulos poderiam explorar integração com bancos de dados, autenticação ou testes automatizados.

Dicas para Continuar

- Adicione um banco de dados como PostgreSQL usando bibliotecas como `pg` (Node.js).
- Implemente autenticação com JWT para proteger endpoints.
- Use ferramentas como **Jest** para testes automatizados.

Conexão com Capítulos Anteriores

- **Capítulo 3:** Os endpoints seguem os princípios REST (interface uniforme, sem estado, HATEOAS).
- **Capítulo 4:** Os payloads JSON respeitam boas práticas (nomes claros, validação, ISO 8601).
- **Capítulo 5:** A API foi expandida a partir dos endpoints GET, mantendo consistência.