

API de Revisão de Endpoints HTTP - Guia Completo

Este projeto é um material didático para aprender desenvolvimento de APIs REST com **Node.js**, **TypeScript** e **Express**. Foi estruturado como uma progressão do básico ao intermediário, com exercícios práticos para consolidar o aprendizado.

Objetivos de Aprendizagem

- **Métodos HTTP:** Entender e implementar os 5 principais verbos HTTP: `GET`, `POST`, `PUT`, `PATCH` e `DELETE`
- **Passagem de Parâmetros:** Dominar as três formas principais: *Route Params*, *Query Params* e *Request Body*
- **Validações:** Implementar validações básicas e robustas de dados
- **Status Codes:** Usar códigos HTTP apropriados para cada situação
- **TypeScript:** Aplicar conceitos básicos de tipagem com TypeScript
- **Boas Práticas:** Estrutura de projeto e respostas HTTP consistentes

Configuração Inicial

Pré-requisitos

- [Node.js](#) (versão 18+)
- [npm](#) ou [yarn](#)
- Editor de código (VS Code recomendado)
- Cliente HTTP (Postman, Thunder Client, Insomnia)

Configuração Inicial

1. **Inicialize o projeto Node.js:**

```
npm init -y
```

2. Instale as dependências necessárias:

```
npm install express typescript @types/express @types/node ts-node-dev
```

3. Inicialize a configuração do TypeScript:

```
npx tsc --init
```

4. Ajuste o arquivo tsconfig.json:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

5. Configure o `package.json`:

Adicione os seguintes scripts:

```
{
  "scripts": {
    "dev": "ts-node-dev --respawn --transpile-only src/server.ts",
    "build": "tsc",
    "start": "node dist/server.js"
  }
}
```




```
}  
}
```

Estrutura do Projeto



```
endpoints-review/  
├── src/  
│   └── server.ts      # Arquivo principal com exercícios  
├── package.json  
├── tsconfig.json  
└── README.md          # Este arquivo
```

Como Usar Este Guia

Legenda dos Símbolos

-  = Método implementado como EXEMPLO (NÃO ALTERAR)
-  = Método para SER IMPLEMENTADO pelos alunos
-  = Seção de conceitos e teoria

Metodologia de Estudo

1. **Leia os conceitos** em cada seção
2. **Estude os exemplos** marcados com 
3. **Implemente os exercícios** marcados com 
4. **Teste cada endpoint** com um cliente HTTP
5. **Compare** sua implementação com os exemplos

Roteiro de Exercícios

Seção 1: Métodos GET - Conceitos Básicos

Conceitos Abordados:

- O que é o método GET
- Route Parameters vs Query Parameters
- Tratamento de erros 404
- Filtros e buscas

✅ Exemplo Implementado

- `GET /users` - Listar todos os usuários
- `GET /users/search` - Busca com filtros avançados

🔧 Exercícios para Implementar

EXERCÍCIO 1: GET com Route Parameter

GET /users/:id

Objetivo: Buscar um usuário específico pelo ID

Dicas de Implementação:

```
// Capturar o ID da URL
const userId = parseInt(req.params.id);

// Buscar usuário
const user = users.find(u => u.id === userId);

// Validar se encontrou
if (!user) {
  return res.status(404).json({
    success: false,
    message: 'Usuário não encontrado'
  });
}
```

Teste:

```
GET <http://localhost:3000/users/1>    # Deve retornar usuário  
GET <http://localhost:3000/users/999>  # Deve retornar 404
```

EXERCÍCIO 2: GET com Query Parameters Avançados

```
GET /users/age-range?min=25&max=35
```

Objetivo: Filtrar usuários por faixa etária

Dicas:

- Capture `req.query.min` e `req.query.max`
- Valide se são números válidos
- Filtre usuários dentro do range
- Retorne erro se parâmetros inválidos

Teste:

```
GET <http://localhost:3000/users/age-range?min=25&max=35>  
GET <http://localhost:3000/users/age-range?min=abc>    # Deve dar erro
```

Seção 2: Métodos POST - Criando Dados

Conceitos Abordados:

- Request Body e JSON
- Validação de dados
- Status 201 (Created)
- Prevenção de duplicatas

Exemplo Implementado

- `POST /users` - Criar usuário com validações completas

Exercícios para Implementar

EXERCÍCIO 3: POST com Validações Personalizadas

```
POST /posts
Content-Type: application/json

{
  "title": "Meu Post",
  "content": "Conteúdo do post...",
  "authorId": 1
}
```

Objetivo: Criar sistema de posts relacionados aos usuários

Regras de Validação:

- `title` : mínimo 3 caracteres
- `content` : mínimo 10 caracteres
- `authorId` : deve existir na lista de usuários
- Posts são criados como `published: false`

Estrutura do Post:

```
interface Post {
  id: number;
  title: string;
  content: string;
  authorId: number;
  createdAt: Date;
  published: boolean;
}
```

Seção 3: Métodos PUT e PATCH - Atualizando Dados

Conceitos Abordados:

- Diferença entre PUT (completo) e PATCH (parcial)
- Spread operator para atualizações

- Validação de campos permitidos
- Preservação de dados importantes (ID, etc.)

✓ Exemplo Implementado

- `PATCH /users/:id` - Atualização parcial avançada

🔧 Exercícios para Implementar

EXERCÍCIO 4: PUT - Atualização Completa

```
PUT /users/1
Content-Type: application/json

{
  "name": "Nome Completo",
  "email": "novo@email.com",
  "role": "admin",
  "age": 30
}
```

Regras para PUT:

- TODOS os campos devem ser fornecidos
- Substituir o objeto completamente
- Validar como no POST
- Verificar conflitos de email

EXERCÍCIO 5: PATCH para Posts

```
PATCH /posts/1
Content-Type: application/json

{
  "title": "Título Atualizado"
}
```

Campos Permitidos: `title` , `content` , `published` **Não Permitir:** `id` , `authorId` , `createdAt`



Seção 4: Métodos DELETE - Removendo Dados

Conceitos Abordados:

- Idempotência do DELETE
- Regras de negócio (não remover último admin)
- Autorização e permissões
- Remoção em lote



Exemplos Implementados

- `DELETE /users/:id` - Remoção com regras de negócio
- `DELETE /users/bulk-delete` - Remoção em lote



Exercícios para Implementar

EXERCÍCIO 6: DELETE com Autorização

```
DELETE /posts/1
User-Id: 1
```

Regras:

- Apenas autor do post ou admins podem remover
- Verificar se post existe
- Para simplificar, usar header `User-Id`

EXERCÍCIO 7: DELETE Condicional

```
DELETE /users/cleanup-inactive?confirm=true
```

Objetivo: Remover usuários sem posts

Regras:

- Não remover administradores

- Parâmetro `confirm=true` obrigatório
- Retornar lista de usuários removidos

Testando os Endpoints

Usando Thunder Client (VS Code)

1. Instale a extensão **Thunder Client**
2. Crie uma nova coleção "API Review"
3. Adicione requests para cada endpoint
4. Teste cenários de sucesso e erro

Exemplos de Requests

Listar Usuários:

```
GET <http://localhost:3000/users>
```

Criar Usuário:

```
POST <http://localhost:3000/users>
```

```
Content-Type: application/json
```

```
{  
  "name": "João Silva",  
  "email": "joao@email.com",  
  "role": "user",  
  "age": 30  
}
```

Buscar com Filtros:

```
GET <http://localhost:3000/users/search?role=admin&name=joão>
```

✓ Lista de Verificação

Básico

- ☐ GET simples funcionando
- ☐ Route parameters implementados
- ☐ Query parameters funcionando
- ☐ POST com validações
- ☐ Tratamento de erros 404/400

Intermediário

- ☐ PUT vs PATCH implementados
- ☐ DELETE com regras de negócio
- ☐ Validações robustas
- ☐ Status codes apropriados
- ☐ Interface ApiResponse consistente

🎓 Dicas de Boas Práticas

1. Validação de Dados

```
// ✓ Bom
if (!name || typeof name !== 'string') {
  errors.push('Nome é obrigatório');
}

// ✗ Evitar
if (!name) {
  return res.status(400).json({ error: 'Nome obrigatório' });
}
```

2. Status Codes Apropriados

- **200** - Sucesso geral
- **201** - Criado (POST)
- **400** - Dados inválidos
- **404** - Não encontrado
- **409** - Conflito (email duplicado)
- **501** - Não implementado (exercícios)

3. Estrutura de Resposta Consistente

```
const response: ApiResponse = {
  success: true,
  message: 'Operação realizada com sucesso',
  data: resultado,
  total: quantidade // quando aplicável
};
```

4. Logs Úteis

```
console.log(`📋 GET /users/${userId} - Buscando usuário por ID`);
console.log('Dados recebidos:', req.body);
```

Problemas Comuns

1. Servidor não inicia

```
# Verifique se as dependências estão instaladas
npm install
```

```
# Verifique se a porta 3000 está livre
npx kill-port 3000
```

2. Erros de TypeScript

```
# Compile manualmente para ver erros
npx tsc --noEmit
```

3. Endpoints não funcionam

- Verifique se o servidor está rodando
- Confirme a URL e método HTTP
- Verifique o Content-Type para POST/PUT/PATCH
- Use `console.log()` para debug

Recursos Adicionais

Documentação

- [Express.js](#)
- [TypeScript](#)
- [HTTP Status Codes](#)

Ferramentas Recomendadas

- **VS Code** com extensões TypeScript e Thunder Client
- **Postman** para testes de API
- **Git** para versionamento

Bom estudo!

Lembre-se: o aprendizado vem da prática. Implemente cada exercício com calma, teste bastante e não tenha medo de experimentar!