📚 API Livraria com Express.js — Parte 5 (Camada Model)

Evoluindo o projeto: adicionando o Model `Livro`

👨‍🏫 **Professor:** Fabricio Bizotto

📘 **Disciplina:** Desenvolvimento Web I

🎓 **Curso:** Ciência da Computação

📅 **Fase:** 4ª fase

Roteiro

- O que é a camada Model
- Implementar `Livro` (validação + serialização)
- Integrar o `Model` no Controller
- Boas práticas e testes rápidos

Estrutura atualizada do projeto

```
livraria_node_http/
├─ server.js
├─ .env
├─ package.json
└─ src/
   ├─ app.js
   ├─ config/express.js
   ├─ data/
   │  └─ livros.json
   ├─ controllers/
   │  └─ livros.controller.js
   ├─ routes/
   │  ├─ index.js
   │  └─ livros.routes.js
   ├─ middlewares/
   │  └─ errorHandler.js
   └─ models/
      └─ livro.model.js   <-- NOVO!
```

Por que um Model?

- Centraliza regras e validações da entidade `Livro`
- Evita lógica duplicada entre rotas e controller
- Facilita testes e evolução (ex.: trocar JSON por DB)
- Garante forma consistente de entrada/saída (toJSON/fromJSON)

src/models/livro.model.js

```
// src/models/livro.model.js
class Livro {
  constructor({ id = null, titulo, autor, categoria, ano }) {
    this.id = id !== undefined ? id : null;
    this.titulo = String(titulo).trim();
    this.autor = String(autor).trim();
    this.categoria = String(categoria).trim();
    this.ano = Number.isInteger(ano) ? ano : parseInt(ano, 10);
    this._validar();
  }
  static fromJSON(json) {
    return new Livro({
      id: json.id ?? null,
      titulo: json.titulo,
      autor: json.autor,
      categoria: json.categoria,
      ano: json.ano,
    });
  }
  toJSON() {
    return {
      id: this.id,
      titulo: this.titulo,
      autor: this.autor,
      categoria: this.categoria,
      ano: this.ano,
    };
  }
  _validar() {
    const erros = [];
    if (!this.titulo || this.titulo.trim().length === 0) erros.push('Título é obrigatório');
    if (!this.autor || this.autor.trim().length === 0) erros.push('Autor é obrigatório');
    if (!this.categoria || this.categoria.trim().length === 0) erros.push('Categoria é obrigatória');
    if (!Number.isInteger(this.ano) || isNaN(this.ano)) erros.push('Ano deve ser um número válido');
    if (erros.length > 0) {
      const error = new Error('Dados inválidos');
      error.statusCode = 400;
      error.details = erros;
      throw error;
    }
  }
}
module.exports = Livro;
```

Integrando o Model no Controller

Exemplo de uso na criação e leitura:

```
// src/controllers/livros.controller.js (trecho)
const Livro = require('../models/livro.model');

async criarLivro(req, res, next) {
  try {
    const { titulo, autor, categoria, ano } = req.body;
    const todos = await this._lerTodosLivros();
    const novoId = todos.length === 0 ? 1 : Math.max(...todos.map(l => l.id)) + 1;
    const livro = new Livro({ id: novoId, titulo, autor, categoria, ano: parseInt(ano, 10) });
    todos.push(livro.toJSON());
    await this._saveToFile(todos);
    res.status(201).json({ mensagem: 'Livro criado', data: livro.toJSON() });
  } catch (err) { next(err); }
}

async listarLivros(req, res, next) {
  try {
    const todos = await this._lerTodosLivros();
    // Normaliza saída via Model (opcional)
    const saida = todos.map(Livro.fromJSON).map(l => l.toJSON());
    res.status(200).json(saida);
  } catch (err) { next(err); }
}
```

Atualizar e Remover com Model

```
async atualizarLivro(req, res, next) {
  try {
    const id = parseInt(req.params.id, 10);
    const { titulo, autor, categoria, ano } = req.body;
    const todos = await this._lerTodosLivros();
    const idx = todos.findIndex(l => l.id === id);
    if (idx === -1) {
      const e = new Error('Livro não encontrado'); e.statusCode = 404; throw e;
    }
    const atualizado = Livro.fromJSON({ ...todos[idx], titulo, autor, categoria, ano: parseInt(ano, 10) }).toJSON();
    todos[idx] = atualizado;
    await this._saveToFile(todos);
    res.status(200).json({ mensagem: 'Atualizado com sucesso', data: atualizado });
  } catch (err) { next(err); }
}

async removerLivro(req, res, next) {
  try {
    const id = parseInt(req.params.id, 10);
    const todos = await this._lerTodosLivros();
    const idx = todos.findIndex(l => l.id === id);
    if (idx === -1) {
      const e = new Error('Livro não encontrado'); e.statusCode = 404; throw e;
    }
    const [removido] = todos.splice(idx, 1);
    await this._saveToFile(todos);
    res.status(200).json({ mensagem: 'Livro removido com sucesso', data: removido });
  } catch (err) { next(err); }
}
```

Boas práticas com Models

- Centralize validações de domínio no Model (ex.: campos obrigatórios, tipos)
- Exponha `fromJSON`/`toJSON` para padronizar conversão
- Não acople o Model a frameworks (Express, DB)
- Use o Model nas entradas/saídas para consistência

Testes rápidos (curl)

```
# Criar (deve validar campos)
curl -X POST http://localhost:3000/api/livros \
  -H "Content-Type: application/json" \
  -d '{"titulo":"DDD","autor":"Evans","categoria":"Arq","ano":2004}'

# Tentar criar com dados inválidos (espera 400)
curl -X POST http://localhost:3000/api/livros \
  -H "Content-Type: application/json" \
  -d '{"titulo":"","autor":"","categoria":"","ano":"abc"}' -i

# Atualizar e listar
curl -X PUT http://localhost:3000/api/livros/1 -H "Content-Type: application/json" \
  -d '{"titulo":"Novo Título","autor":"Autor","categoria":"Cat","ano":2025}'

curl http://localhost:3000/api/livros
```

Próximos passos

- Separar persistência em Repository (se necessário escalar)
- Introduzir banco (SQLite/Postgres) e ORM leve (Prisma/Knex)
- Escrever testes automatizados (Jest + supertest)
- Versionar API e documentar com OpenAPI/Swagger

Encerramento

- Model traz robustez e consistência às entidades
- Com ele, a API fica mais previsível e fácil de evoluir