


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

Model-View-Controller (MVC)

+-----+	
Model	<--- Regras de negócio, validação, estrutura de dados
+-----+	
View	<--- Apresentação dos dados (HTML, JSON, etc.)
+-----+	
Controller	<--- Lógica de controle, intermediação entre Model e View
+-----+	

O Model representa a estrutura e as regras dos dados da aplicação.

Estrutura atualizada do projeto

```
livraria_node_http/  
├── server.js  
├── .env  
├── package.json  
└── src/  
    ├── app.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();
  }

  _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;
```

src/models/livro.model.js (continuação...)

```
// src/models/livro.model.js (continuação)
class Livro {

  // ...

  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,
    };
  }
}

module.exports = Livro;
```

Integrando o Model ao Repository

```
// src/repositories/livros.repository.js
const Livro = require("../models/livro.model");

class LivrosRepository extends RepositoryBase {

  async findAll() {
    const dados = await this._lerArquivo();
    const lista = JSON.parse(dados);
    return lista.map(item => Livro.fromJSON(item)); // <<< ALTERAÇÃO
  }

  // async findById(id) não precisa mudar

  async create(livroData) {
    const livros = await this.findAll();
    const novoId = await this.getNextId();
    const novoLivro = new Livro({ id: novoId, ...livroData }); // <<< ALTERAÇÃO
    livros.push(novoLivro);
    await this._saveToFile(livros.map(l => l.toJSON())); // <<< ALTERAÇÃO
    return novoLivro;
  }
}

module.exports = LivrosRepository;
```


Integrando o Model ao Controller (continuação...)

```
// src/repositories/livros.repository.js (continuação)

async update(id, dadosAtualizados) {
  const livros = await this.findAll();
  const indice = livros.findIndex(livro => livro.id === id);

  if (indice === -1) {
    const error = new Error("Livro não encontrado");
    error.statusCode = 404;
    throw error;
  }
  livros[indice] = new Livro({ ...livros[indice], ...dadosAtualizados }); // <<< ALTERAÇÃO
  await this._saveToFile(livros.map(l => l.toJSON())); // <<< ALTERAÇÃO
  return livros[indice];
}

async delete(id) {
  const livros = await this.findAll();
  const indice = livros.findIndex(livro => livro.id === id);

  if (indice === -1) {
    const error = new Error("Livro não encontrado");
    error.statusCode = 404;
    throw error;
  }

  const [livroRemovido] = livros.splice(indice, 1);
  await this._saveToFile(livros.map(l => l.toJSON())); // <<< ALTERAÇÃO
  return livroRemovido;
}
```

Ajustes nas Rotas

Podemos fazer `bind` para manter o contexto do `this`:

```
// src/routes/livros.routes.js
router.get("/", livrosController.listarLivros.bind(livrosController));
router.get("/:id", validarParamId, livrosController.buscarLivroPorId.bind(livrosController));
router.post("/", validarLivro, livrosController.criarLivro.bind(livrosController));
router.put("/:id", validarParamId, validarLivro, livrosController.atualizarLivro.bind(livrosController));
router.delete("/:id", validarParamId, livrosController.removerLivro.bind(livrosController));
```

- O `bind` vincula o contexto do `this` ao controller, garantindo que os métodos funcionem corretamente quando chamados como callbacks. Isso é especialmente útil em rotas, onde o contexto pode ser perdido.
- Podemos também usar arrow functions, mas o `bind` é mais direto e limpo nesse caso.
- Podemos também remover o middleware `validarLivro`, já que a validação agora está no Model.

Próximos passos

- Introduzir banco (SQLite/Postgres) e ORM (Prisma/Knex)
- Escrever testes automatizados
- Versionar API e documentar com OpenAPI/Swagger/Postman

Exercícios/Desafios

- Adicionar os seguintes campos ao Model `Livro`:
 - `editora` (string, opcional)
 - `paginas` (número inteiro positivo, opcional)
- Atualizar o repositório e controller para suportar os novos campos.