

# Construindo uma API REST Spring Boot Java para Controle de Estoque

Prof. Leonardo Arruda  
Prof. Rafael Cruz  
2025

## Sumário

Introdução	3
Objetivos:	3
Passo 1: Configuração do Projeto (Spring Initializr)	4
Passo 2: Configuração do Banco de Dados	4
Passo 3: Criação das Entidades (Modelos JPA)	5
1. Entidade Categoria (Lado 1:N)	5
2. Entidade Fornecedor (Lado N:M)	6
3. Entidade Estoque (Relacionamento 1:1)	7
4. Entidade Produto (Relacionamentos 1:1, N:1, N:M)	8
Passo 4: Notações JPA/Hibernate (Resumo)	9
Passo 5: Criação dos Repositórios	10
Passo 6: Criação dos Controllers REST	11
1. CategoriaController	12
2. FornecedorController	13
3. ProdutoController (Gerencia 1:1, N:1, N:M)	14
Como testar (Exemplo de JSON)	16
Conclusão	16
Referências Bibliográficas	16

## Introdução

Este guia detalha o desenvolvimento de uma API REST para Controle de Estoque utilizando Spring Boot 3 e Spring Data JPA, com foco na persistência de dados em um banco de dados MySQL/MariaDB.

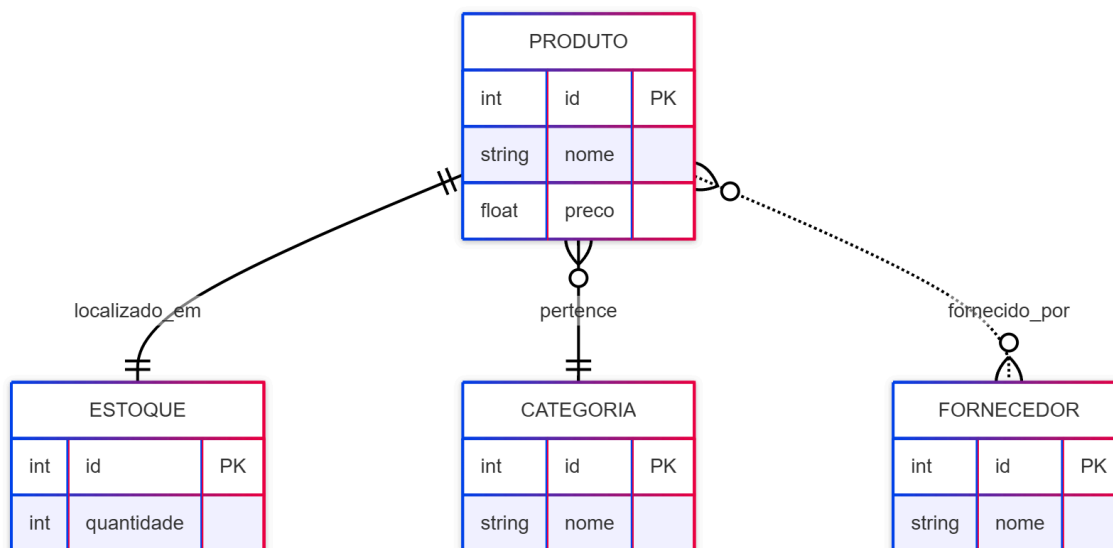


Figura 1 - DER representando o sistema de controle de estoque.

O projeto serve como um estudo de caso prático, demonstrando a arquitetura completa de uma aplicação back-end: desde a configuração inicial do banco de dados e a modelagem das entidades até a criação de endpoints RESTful para manipulação de dados.

## Objetivos:

Os principais objetivos deste guia são:

1. Demonstrar Relacionamentos JPA: Implementar e explicar o mapeamento dos três tipos fundamentais de relacionamento do JPA/Hibernate:
  - Um-para-Um (1:1): Entre Produto e Estoque.
  - Um-para-Muitos (1:N): Entre Categoria e Produto.
  - Muitos-para-Muitos (N:M): Entre Produto e Fornecedor.
2. Configuração de Persistência: Configurar o Spring Boot para se conectar ao MySQL/MariaDB, incluindo o uso correto do dialeto Hibernate e das propriedades de DDL (Data Definition Language).
3. Desenvolvimento da Camada Controller: Criar Controllers RESTful com as anotações corretas (@RestController, @RequestMapping, @PostMapping, etc.) para expor o CRUD (Create, Read, Update, Delete) das entidades.

4. Melhores Práticas de Mapeamento: Aplicar o conceito de criação em cascata (CascadeType.ALL) e o uso de chaves estrangeiras (@JoinColumn) e tabelas de junção (@JoinTable) de forma eficiente e limpa.

## Passo 1: Configuração do Projeto (Spring Initializr)

Configurações iniciais:

- Project: Maven Project
- Language: Java (21 ou 17)
- Spring Boot: 3.2.x (ou mais recente)
- Group: com.controleestoque
- Artifact: api-estoque
- Dependências:
  - o Spring Web
  - o Spring Data JPA
  - o MySQL Driver
  - o Lombok

## Passo 2: Configuração do Banco de Dados

Arquivo: src/main/resources/application.properties

```
# Nome da Aplicação
spring.application.name=api-estoque

# Configurações do Banco de Dados (MySQL/MariaDB)
# ATENÇÃO: Altere 'senha_do_seu_banco' e 'estoque_db' conforme necessário.
spring.datasource.url=jdbc:mysql://localhost:3306/estoque_db?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
# Driver class para MySQL/MariaDB
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Configurações do JPA e Hibernate
# 'update': Tenta criar tabelas se não existirem. Não recomendado em produção!
spring.jpa.hibernate.ddl-auto=update
# Mostra o SQL gerado pelo Hibernate no console
spring.jpa.show-sql=true

# === Configuração do Dialeto ===
# Se estiver usando MariaDB:
# spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect
# Se estiver usando MySQL 8+:
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

# Configur
```

## Passo 3: Criação das Entidades (Modelos JPA)

### 1. Entidade Categoria (Lado 1:N)

Caminho: com.controleestoque.api\_estoque.model.Categoria

```
1 package com.controleestoque.api_estoque.model;
2
3 import jakarta.persistence.*;
4 import java.util.List;
5
6 @Entity
7 @Table(name = "tb_categorias")
8 public class Categoria {
9
10     @Id
11     @GeneratedValue(strategy = GenerationType.IDENTITY)
12     private Long id;
13
14     private String nome;
15
16     // --- Relacionamento 1:N (One-to-Many) ---
17     // É o lado '1' do relacionamento. 'mappedBy' aponta para o campo em Produto.
18     @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)
19     private List<Produto> produtos;
20
21     // Construtores, Getters e Setters...
22     public Categoria() {}
23
24     public Categoria(String nome, List<Produto> produtos) {
25         this.nome = nome;
26         this.produtos = produtos;
27     }
28
29     public Long getId() { return id; }
30     public void setId(Long id) { this.id = id; }
31     public String getNome() { return nome; }
32     public void setNome(String nome) { this.nome = nome; }
33     public List<Produto> getProdutos() { return produtos; }
34     public void setProdutos(List<Produto> produtos) { this.produtos = produtos; }
35
36 }
37
```

## 2. Entidade Fornecedor (Lado N:M)

Caminho: com.controleestoque.api\_estoque.model.Fornecedor

```
1 package com.controleestoque.api_estoque.model;
2
3 import jakarta.persistence.*;
4 import java.util.Set;
5
6 @Entity
7 @Table(name = "tb_fornecedores")
8 public class Fornecedor {
9
10     @Id
11     @GeneratedValue(strategy = GenerationType.IDENTITY)
12     private Long id;
13
14     private String nome;
15
16     // --- Relacionamento N:M (Many-to-Many) ---
17     // Mapeamento: Lado inverso do relacionamento em Produto.
18     // 'mappedBy' indica que o mapeamento da tabela de junção está na classe Produto.
19     @ManyToMany(mappedBy = "fornecedores")
20     private Set<Produto> produtos;
21
22     // Construtores, Getters e Setters...
23     public Fornecedor() {}
24
25     public Fornecedor(String nome, Set<Produto> produtos) {
26         this.nome = nome;
27         this.produtos = produtos;
28     }
29
30     public Long getId() { return id; }
31     public void setId(Long id) { this.id = id; }
32     public String getNome() { return nome; }
33     public void setNome(String nome) { this.nome = nome; }
34     public Set<Produto> getProdutos() { return produtos; }
35     public void setProdutos(Set<Produto> produtos) { this.produtos = produtos; }
36
37 }
38
```

### 3. Entidade Estoque (Relacionamento 1:1)

Caminho: com.controleestoque.apiestoque.model.Estoque

```
1 package com.controleestoque.api_estoque.model;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 @Table(name = "tb_estoques")
7 public class Estoque {
8
9     @Id
10    @GeneratedValue(strategy = GenerationType.IDENTITY)
11    private Long id;
12
13    private Integer quantidade;
14
15    // --- Relacionamento 1:1 (One-to-One) ---
16    // É o lado 'proprietário' que contém a chave estrangeira (FK).
17    @OneToOne(fetch = FetchType.LAZY)
18    @JoinColumn(name = "produto_id", nullable = false) // Define a FK na tabela tb_estoques.
19    private Produto produto;
20
21    // Construtores, Getters e Setters...
22    public Estoque() {}
23
24    public Estoque(Integer quantidade, Produto produto) {
25        this.quantidade = quantidade;
26        this.produto = produto;
27    }
28
29    public Long getId() { return id; }
30    public void setId(Long id) { this.id = id; }
31    public Integer getQuantidade() { return quantidade; }
32    public void setQuantidade(Integer quantidade) { this.quantidade = quantidade; }
33    public Produto getProduto() { return produto; }
34    public void setProduto(Produto produto) { this.produto = produto; }
35
36 }
37
```

## 4. Entidade Produto (Relacionamentos 1:1, N:1, N:M)

Caminho: com.controleestoque.api\_estoque.model.Produto

```
1 package com.controleestoque.api_estoque.model;
2
3 import java.math.BigDecimal;
4 import java.util.Set;
5
6 import jakarta.persistence.*;
7
8 @Entity
9 @Table(name = "tb_produtos")
10 public class Produto {
11
12     @Id
13     @GeneratedValue(strategy = GenerationType.IDENTITY)
14     private Long id; // Chave primária.
15
16     private String nome;
17
18     private BigDecimal preco;
19
20     // --- Relacionamento 1:1 (One-to-One) ---
21     // Mapeamento: Um produto tem UM registro de estoque (e vice-versa).
22     // 'mappedBy' indica que a chave estrangeira está na classe Estoque.
23     // Cascade.ALL: Operações (como salvar) em Produto afetam Estoque.
24     @OneToOne(mappedBy = "produto", cascade = CascadeType.ALL, orphanRemoval = true)
25     private Estoque estoque;
26
27     // --- Relacionamento N:1 (Many-to-One) ---
28     // Mapeamento: Muitos produtos têm UMA categoria.
29     // É o lado 'N' (Muitos), que contém a chave estrangeira (FK).
30     @ManyToOne(fetch = FetchType.LAZY) // LAZY: Carrega a categoria apenas quando for solicitada.
31     @JoinColumn(name = "categoria_id", nullable = false) // Define a FK na tabela tb_produtos.
32     private Categoria categoria;
33
34     // --- Relacionamento N:M (Many-to-Many) ---
35     // Mapeamento: Muitos produtos têm MUITOS fornecedores (e vice-versa).
36     // Define a tabela intermediária tb_produto_fornecedor e as colunas de união.
37     @ManyToMany
38     @JoinTable(
39         name = "tb_produto_fornecedor", // Nome da tabela de junção
40         joinColumns = @JoinColumn(name = "produto_id"), // FK desta entidade na tabela de junção
41         inverseJoinColumns = @JoinColumn(name = "fornecedor_id") // FK da outra entidade
42     )
43     private Set<Fornecedor> fornecedores;
44
45     // Construtores, Getters e Setters...
46     public Produto(){}
47
48     public Produto(String nome, BigDecimal preco, Estoque estoque, Categoria categoria,
49         Set<Fornecedor> fornecedores) {
50         this.nome = nome;
51         this.preco = preco;
52         this.estoque = estoque;
53         this.categoria = categoria;
54         this.fornecedores = fornecedores;
55     }
56
57     public Long getId() { return id; }
58     public void setId(Long id) { this.id = id; }
59     public String getNome() { return nome; }
60     public void setNome(String nome) { this.nome = nome; }
61     public BigDecimal getPreco() { return preco; }
62     public void setPreco(BigDecimal preco) { this.preco = preco; }
63     public Estoque getEstoque() { return estoque; }
64     public void setEstoque(Estoque estoque) { this.estoque = estoque; }
65     public Categoria getCategoria() { return categoria; }
66     public void setCategoria(Categoria categoria) { this.categoria = categoria; }
67     public Set<Fornecedor> getFornecedores() { return fornecedores; }
68     public void setFornecedores(Set<Fornecedor> fornecedores) { this.fornecedores = fornecedores; }
69
70 }
71
72
```

## Passo 4: Notações JPA/Hibernate (Resumo)

As anotações principais que definem os relacionamentos e a estrutura do banco de dados são:

Anotação	Tipo	Uso
<b>@Entity</b>	JPA	Marca a classe como uma entidade persistente no banco de dados.
<b>@Table</b>	JPA	Especifica o nome da tabela no banco de dados.
<b>@Id</b>	JPA	Marca o campo como a chave primária da entidade.
<b>@GeneratedValue</b>	JPA	Especifica como a chave primária é gerada (ex: GenerationType.IDENTITY usa auto-incremento do DB).
<b>@Column</b>	JPA	Define detalhes da coluna (ex: nullable = false, unique = true).
<b>@OneToOne</b>	JPA	Relacionamento 1:1.
<b>@OneToMany</b>	JPA	Relacionamento 1:N (lado do '1').
<b>@ManyToOne</b>	JPA	Relacionamento N:1 (lado do 'N', onde fica a FK).
<b>@ManyToMany</b>	JPA	Relacionamento N:M.
<b>@JoinColumn</b>	JPA	Usada no lado proprietário (onde está a FK) para definir o nome da coluna de chave estrangeira (ex: categoria_id).
<b>@JoinTable</b>	JPA	Usada no relacionamento N:M para definir a tabela de junção intermediária.
<b>mappedBy</b>	JPA	Usada no lado inverso do relacionamento para indicar qual campo na classe proprietária gerencia o mapeamento.

## Passo 5: Criação dos Repositórios

A camada Repository (ou Camada de Persistência) é responsável pela **comunicação direta com o banco de dados**. Utilizando o **Spring Data JPA**, esta camada abstrai a complexidade do SQL, permitindo que o desenvolvedor trabalhe com métodos de alto nível.

Característica	Função no Projeto
<b>Anotação Chave</b>	@Repository (Implícita ao estender JpaRepository)
<b>Responsabilidade</b>	<b>CRUD:</b> Implementa o Create, Read, Update e Delete para as entidades. <b>Transações:</b> Garante que as operações de banco de dados sejam atômicas. <b>Mapeamento:</b> Converte objetos Java (Produto) para registros do banco de dados (tabela tb_produtos) e vice-versa.
<b>Fluxo de Dados</b>	Recebe objetos da camada superior $\rightarrow$ Traduz para comandos SQL/JPA $\rightarrow$ Interage com o MySQL/MariaDB.
<b>Objetivo</b>	Isolar toda a lógica de acesso ao banco de dados, permitindo que o restante da aplicação não precise saber como os dados são armazenados ou recuperados.

Crie interfaces para cada entidade estendendo JpaRepository (ex: ProdutoRepository, CategoriaRepository, etc.).

```
1 package com.controleestoque.api_estoque.repository;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import org.springframework.stereotype.Repository;
5 import com.controleestoque.api_estoque.model.Produto;
6
7 @Repository
8 public interface ProdutoRepository extends JpaRepository<Produto, Long>{
9     // Métodos personalizados podem ser adicionados aqui (ex: findByNome)
10 }
```

## Passo 6: Criação dos Controllers REST

A camada Controller atua como o **ponto de entrada da sua API**. Ela é responsável por receber e rotear requisições HTTP do cliente (navegador, Postman, aplicação front-end) e por formatar a resposta a ser enviada de volta.

Característica	Função no Projeto
<b>Anotações Chave</b>	@RestController, @RequestMapping, @GetMapping, @PostMapping, etc.
<b>Responsabilidade</b>	<b>Receber Requisições:</b> Mapeia URLs para métodos Java. <b>Validação Básica:</b> Verifica se o corpo da requisição (@RequestBody) está presente. <b>Resposta HTTP:</b> Define o status da resposta (ex: 200 OK, 201 Created, 404 Not Found).
<b>Fluxo de Dados</b>	Recebe dados do cliente (JSON) → Desencapsula → Chama a camada de Serviço (ou Repository, neste caso simples).
<b>Objetivo</b>	Garantir que a comunicação com o mundo externo (HTTP) seja tratada corretamente, mantendo a lógica de negócio <b>fora</b> desta camada.

## 1. CategoriaController

Caminho: com.controleestoque.apiestoque.controller.CategoriaController

```
1 package com.controleestoque.api_estoque.controller;
2
3 import com.controleestoque.api_estoque.model.Categoria;
4 import com.controleestoque.api_estoque.repository.CategoriaRepository;
5 import lombok.RequiredArgsConstructor;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.http.ResponseEntity;
8 import org.springframework.web.bind.annotation.*;
9
10 import java.util.List;
11
12 @RestController
13 @RequestMapping("/api/categorias") // Define o caminho base para este controller
14 @RequiredArgsConstructor // Injeta automaticamente o CategoriaRepository via construtor
15 public class CategoriaController {
16
17     private final CategoriaRepository categoriaRepository;
18
19     // GET /api/categorias
20     @GetMapping
21     public List<Categoria> getAllCategorias() {
22         // Retorna todas as categorias do banco de dados
23         return categoriaRepository.findAll();
24     }
25
26     // GET /api/categorias/{id}
27     @GetMapping("/{id}")
28     public ResponseEntity<Categoria> getCategoriaById(@PathVariable Long id) {
29         // Busca a categoria pelo ID. UsaorElse para retornar 404 se não encontrar.
30         return categoriaRepository.findById(id)
31             .map(ResponseEntity::ok)
32             .orElse(ResponseEntity.notFound().build());
33     }
34
35     // POST /api/categorias
36     @PostMapping
37     @ResponseStatus(HttpStatus.CREATED) // Retorna código 201 (Created)
38     public Categoria createCategoria(@RequestBody Categoria categoria) {
39         // Salva uma nova categoria no banco de dados
40         return categoriaRepository.save(categoria);
41     }
42
43     // PUT /api/categorias/{id}
44     @PutMapping("/{id}")
45     public ResponseEntity<Categoria> updateCategoria(
46         @PathVariable Long id, @RequestBody Categoria categoriaDetails) {
47         // Tenta encontrar a categoria existente
48         return categoriaRepository.findById(id)
49             .map(categoria -> {
50                 // Atualiza os dados da categoria encontrada
51                 categoria.setNome(categoriaDetails.getNome());
52                 Categoria updatedCategoria = categoriaRepository.save(categoria);
53                 return ResponseEntity.ok(updatedCategoria);
54             })
55             .orElse(ResponseEntity.notFound().build());
56     }
57
58     // DELETE /api/categorias/{id}
59     @DeleteMapping("/{id}")
60     public ResponseEntity<Void> deleteCategoria(@PathVariable Long id) {
61         // Tenta encontrar e deletar
62         if (!categoriaRepository.existsById(id)) {
63             return ResponseEntity.notFound().build();
64         }
65         categoriaRepository.deleteById(id);
66         return ResponseEntity.noContent().build(); // Retorna código 204 (No Content)
67     }
68 }
69
```

## 2. FornecedorController

Caminho: com.controleestoque.apiestoque.controller.FornecedorController

```
1 package com.controleestoque.api_estoque.controller;
2
3 import java.util.List;
4
5 import org.springframework.http.*;
6 import org.springframework.web.bind.annotation.*;
7
8 import com.controleestoque.api_estoque.model.Fornecedor;
9 import com.controleestoque.api_estoque.repository.FornecedorRepository;
10
11 import lombok.RequiredArgsConstructor;
12
13
14 @RestController
15 @RequestMapping("/api/fornecedores")
16 @RequiredArgsConstructor
17 public class FornecedorController {
18
19     private final FornecedorRepository fornecedorRepository;
20
21     @GetMapping
22     public List<Fornecedor> getAllFornecedores() {
23         return fornecedorRepository.findAll();
24     }
25
26     // GET /api/fornecedores/{id}
27     @GetMapping("/{id}")
28     public ResponseEntity<Fornecedor> getCategoriaById(@PathVariable Long id) {
29         // Busca a fornecedor pelo ID. Usa orElse para retornar 404 se não encontrar.
30         return fornecedorRepository.findById(id)
31             .map(ResponseEntity::ok)
32             .orElse(ResponseEntity.notFound().build());
33     }
34
35     @PostMapping
36     @ResponseStatus(HttpStatus.CREATED)
37     public Fornecedor createFornecedor(@RequestBody Fornecedor fornecedor) {
38         return fornecedorRepository.save(fornecedor);
39     }
40
41     // PUT /api/fornecedor/{id}
42     @PutMapping("/{id}")
43     public ResponseEntity<Fornecedor> updateFornecedor(
44         @PathVariable Long id, @RequestBody Fornecedor fornecedorDetails) {
45         // Tenta encontrar o fornecedor existente
46         return fornecedorRepository.findById(id)
47             .map(fornecedor -> {
48                 // Atualiza os dados do fornecedor encontrada
49                 fornecedor.setNome(fornecedorDetails.getNome());
50                 Fornecedor updatedFornecedor = fornecedorRepository.save(fornecedor);
51                 return ResponseEntity.ok(updatedFornecedor);
52             })
53             .orElse(ResponseEntity.notFound().build());
54     }
55
56     // DELETE /api/fornecedor/{id}
57     @DeleteMapping("/{id}")
58     public ResponseEntity<Void> deleteFornecedor(@PathVariable Long id) {
59         // Tenta encontrar e deletar
60         if (!fornecedorRepository.existsById(id)) {
61             return ResponseEntity.notFound().build();
62         }
63         fornecedorRepository.deleteById(id);
64         return ResponseEntity.noContent().build(); // Retorna código 204 (No Content)
65     }
66 }
67
```

### 3. ProdutoController (Gerencia 1:1, N:1, N:M)

Caminho: com.controleestoque.apiestoque.controller.ProdutoController

```
package com.controleestoque.api_estoque.controller;

import java.util.List;

import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;

import com.controleestoque.api_estoque.model.Produto;
import com.controleestoque.api_estoque.repository.ProdutoRepository;

import lombok.RequiredArgsConstructor;

import com.controleestoque.api_estoque.repository.CategoriaRepository;
import com.controleestoque.api_estoque.repository.FornecedorRepository;

@RestController
@RequestMapping("/api/produtos")
@RequiredArgsConstructor
public class ProdutoController {

    private final ProdutoRepository produtoRepository;
    private final CategoriaRepository categoriaRepository;
    private final FornecedorRepository fornecedorRepository;
    // Estoque é geralmente manipulado via Produto ou separadamente.

    // GET /api/produtos
    @GetMapping
    public List<Produto> getAllProdutos() {
        // Retorna a lista de produtos. Pode ser necessário configurar DTOs para evitar loops infinitos com JSON.
        return produtoRepository.findAll();
    }

    // GET /api/produtos/{id}
    @GetMapping("/{id}")
    public ResponseEntity<Produto> getCategoriaById(@PathVariable Long id) {
        // Busca a categoria pelo ID. Usa orElse para retornar 404 se não encontrar.
        return produtoRepository.findById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}
```

```
// POST /api/produtos
// Neste método, assumimos que a Categoria e os Fornecedores já existem
// e seus IDs são passados no corpo da requisição (ProdutoDTO seria o ideal aqui).
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<Produto> createProduto(@RequestBody Produto produto) {
    // 1. Gerenciamento do 1:N (Categoria)
    // A categoria deve ser buscada para garantir que existe e estar no contexto de persistência.
    if (produto.getCategoria() == null || produto.getCategoria().getId() == null) {
        return ResponseEntity.badRequest().build(); // Categoria é obrigatória
    }
    categoriaRepository.findById(produto.getCategoria().getId())
        .ifPresent(produto::setCategoria); // Associa a categoria gerenciada

    // 2. Gerenciamento do N:M (Fornecedores)
    // Busca todos os fornecedores pelos IDs fornecidos
    if (produto.getFornecedores() != null && !produto.getFornecedores().isEmpty()) {
        // Cria um Set para armazenar os fornecedores gerenciados
        produto.getFornecedores().clear();

        // Aqui, em um projeto real, você buscaria os Fornecedores um por um
        // ou usando um método customizado do repositório.
        // Exemplo Simplificado:
        produto.getFornecedores().forEach(fornecedor -> {
            fornecedorRepository.findById(fornecedor.getId())
                .ifPresent(produto.getFornecedores()::add); // Adiciona o fornecedor gerenciado
        });
    }

    // 3. Salva o Produto (e o Estoque, se o CASCADE estiver configurado)
    Produto savedProduto = produtoRepository.save(produto);

    return ResponseEntity.status(HttpStatus.CREATED).body(savedProduto);
}

// PUT /api/produtos/{id}
@PutMapping("/{id}")
public ResponseEntity<Produto> updateProduto(@PathVariable Long id, @RequestBody Produto produtoDetails) {
    // Tenta encontrar o produto existente
    return produtoRepository.findById(id)
        .map(produto -> {
            // Atualiza os dados do produto encontrada
            produto.setNome(produtoDetails.getNome());
            Produto updatedProduto = produtoRepository.save(produto);
            return ResponseEntity.ok(updatedProduto);
        })
        .orElse(ResponseEntity.notFound().build());
}

// DELETE /api/produtos/{id}
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteProduto(@PathVariable Long id) {
    // Tenta encontrar e deletar
    if (!produtoRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    produtoRepository.deleteById(id);
    return ResponseEntity.noContent().build(); // Retorna código 204 (No Content)
}
}
```

## Como testar (Exemplo de JSON)

**Requisição POST** para `http://localhost:8080/api/produtos`:

```
{
  "nome": "Notebook Gamer Pro",
  "preco": 4500.00,

  "categoria": {
    "id": 1
  },

  "fornecedores": [
    { "id": 1 }
  ],

  "estoque": {
    "quantidade": 15
  }
}
```

## Conclusão

Este guia demonstrou a construção de uma **API REST funcional para Controle de Estoque** baseada na arquitetura **Spring Boot e Spring Data JPA**. Através da implementação das entidades Produto, Categoria, Estoque e Fornecedor, alcançamos o objetivo central de mapear e gerenciar com sucesso os três principais tipos de relacionamento em um banco de dados relacional: Um-para-Um ( $\text{\$}\text{1:1}\text{\$}$ ), Um-para-Muitos ( $\text{\$}\text{1:N}\text{\$}$ ) e Muitos-para-Muitos ( $\text{\$}\text{N:M}\text{\$}$ ).

As camadas **Controller** e **Repository** foram estabelecidas, garantindo a separação de preocupações essenciais para a escalabilidade e manutenção do código. A utilização do mapeamento por cascata na entidade Produto simplificou o gerenciamento do registro de Estoque ( $\text{\$}\text{1:1}\text{\$}$ ), exemplificando uma boa prática de persistência. O projeto está agora estruturalmente pronto para receber a camada de **Serviço (Service Layer)**, onde a lógica de negócio complexa será isolada e implementada de forma robusta.

## Referências Bibliográficas

MACEDO, Alberto; SANTOS, Maurício. **Spring Boot: Acelere o desenvolvimento de microsserviços e APIs com Java e Spring**. 2. ed. São Paulo: Novatec, 2023.

SILVA, Fábio A. da; VIEIRA, J. P. **JPA e Hibernate: Persistência de dados com Java**. Rio de Janeiro: Alta Books, 2021.

GUIMARÃES, Rodrigo; GOMES, Paulo. **REST e RESTful Web Services: Guia para Desenvolvimento com Spring Framework e Spring Boot**. São Paulo: Editora Érica, 2020.

SPRING. **Spring Data JPA**. Disponível em:  
<https://spring.io/projects/spring-data-jpa/>. Acesso em: [3 Nov. 2025].

BÓSON TREINAMENTOS. **Tutorial MySQL**. Disponível em:  
<https://www.bosontreinamentos.com.br/mysql/>. Acesso em: [3 Nov. 2025].

ORACLE. **Java Persistence API (JPA) Specification**. Disponível em:  
<https://jakarta.ee/specifications/persistence/>. Acesso em: [3 Nov. 2025].