Guia Passo a Passo: Implementação de CRUD Completo com Spring Boot (Nível Júnior)

Este guia detalhado é projetado para iniciantes e cobre a implementação de uma API REST de cadastro de produtos usando Spring Boot, conforme os requisitos do teste prático.

1. Configuração Inicial do Projeto

O primeiro passo é criar a estrutura básica do projeto Spring Boot.

1.1. Usando o Spring Initializr

A maneira mais fácil de começar é usando o Spring Initializr.

Campo	Valor Sugerido	Descrição
Project	Maven Project	Padrão e robusto.
Language	Java	Linguagem principal.
Spring Boot	Versão estável mais recente (ex: 3.x)	Escolha uma versão estável.
Group	`com.exemplo`	Identificador da sua organização.
Artifact	`crud-produtos`	Nome do seu projeto.
Name	`crud-produtos`	Nome de exibição.
Package name	`com.exemplo.crudprodutos`	Pacote base.
Packaging	Jar	Padrão para aplicações Spring Boot.
Java	17 ou 21	Versão LTS (Long-Term Support) recomendada.

1.2. Adicionando Dependências (Dependencies)

Adicione as seguintes dependências no Initializr:

1. **Spring Web**: Para construir aplicações web, incluindo APIs RESTful.
2. **Spring Data JPA**: Para persistência de dados usando o padrão JPA (Java Persistence API).
3. **H2 Database**: Um banco de dados em memória simples, ideal para desenvolvimento e testes.
4. **Lombok**: Reduz a verbosidade do código Java (opcional, mas altamente recomendado).
5. **Spring Boot DevTools**: Para recarregamento automático da aplicação durante o desenvolvimento.
6. **Springdoc OpenAPI Starter WebMVC UI**: Para gerar a documentação da API com Swagger.

Após configurar, clique em **Generate** e baixe o arquivo ZIP. Descompacte-o e abra o projeto em sua IDE (IntelliJ IDEA, VS Code, Eclipse).

2. Camada de Modelo (Model/Entity)

Crie a classe que representa o produto e será mapeada para o banco de dados.

2.1. Criando a Entidade `Produto`

Crie o arquivo `Produto.java` no pacote `com.exemplo.crudprodutos.model`.

Java

```java
// src/main/java/com/exemplo/crudprodutos/model/Produto.java
package com.exemplo.crudprodutos.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.PositiveOrZero;
import lombok.Data;

@Entity
@Data // Gera Getters, Setters, toString, equals e hashCode (Lombok)
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @NotBlank(message = "O nome é obrigatório")
    private String nome;

    @NotBlank(message = "A descrição é obrigatória")
    private String descricao;

    @NotNull(message = "O preço é obrigatório")
    @PositiveOrZero(message = "O preço deve ser positivo ou zero")
    private Double preco;

    @NotNull(message = "A quantidade em estoque é obrigatória")
    @PositiveOrZero(message = "A quantidade deve ser positiva ou zero")
    private Integer quantidadeEstoque;

    // Construtores omitidos pelo @Data do Lombok
}
```

Observações Importantes:

- `@Entity`: Marca a classe como uma entidade JPA.
- `@Id` e `@GeneratedValue`: Define a chave primária e a estratégia de geração automática.
- `@Data`: Anotação do Lombok para reduzir o código boilerplate.
- `@NotBlank`, `@NotNull`, `@PositiveOrZero`: **Validações** de campo obrigatórias, conforme o requisito 6.

3. Camada de Persistência (Repository)

Crie a interface que estende o `JpaRepository` para as operações de banco de dados.

3.1. Criando o `ProdutoRepository`

Crie o arquivo `ProdutoRepository.java` no pacote `com.exemplo.crudprodutos.repository`.

Java

```
// src/main/java/com/exemplo/crudprodutos/repository/ProdutoRepository.java
package com.exemplo.crudprodutos.repository;

import com.exemplo.crudprodutos.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    // O JpaRepository já fornece todos os métodos CRUD básicos (save,
```

```
findAll, findById, deleteById)
}
```

4. Camada de Serviço (Service)

Crie a classe de serviço para implementar a lógica de negócio e orquestrar as operações do repositório.

4.1. Criando o `ProdutoService`

Crie o arquivo `ProdutoService.java` no pacote `com.exemplo.crudprodutos.service`.

Java

```java
// src/main/java/com/exemplo/crudprodutos/service/ProdutoService.java
package com.exemplo.crudprodutos.service;

import com.exemplo.crudprodutos.model.Produto;
import com.exemplo.crudprodutos.repository.ProdutoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    // 1. POST /produtos - Cadastrar novo produto
    public Produto salvar(Produto produto) {
        return produtoRepository.save(produto);
    }

    // 2. GET /produtos - Listar todos os produtos
    public List<Produto> listarTodos() {
        return produtoRepository.findAll();
    }

    // 3. GET /produtos/{id} - Buscar produto por ID
    public Optional<Produto> buscarPorId(Long id) {
        return produtoRepository.findById(id);
    }

    // 4. PUT /produtos/{id} - Atualizar produto
```

```
    public Produto atualizar(Long id, Produto produtoDetalhes) {
        return produtoRepository.findById(id).map(produtoExistente -> {
            produtoExistente.setNome(produtoDetalhes.getNome());
            produtoExistente.setDescricao(produtoDetalhes.getDescricao());
            produtoExistente.setPreco(produtoDetalhes.getPreco());

produtoExistente.setQuantidadeEstoque(produtoDetalhes.getQuantidadeEstoque())
;
            return produtoRepository.save(produtoExistente);
        }).orElse(null); // Retorna null se não encontrar, o Controller deve
tratar isso.
    }

    // 5. DELETE /produtos/{id} - Excluir produto
    public boolean excluir(Long id) {
        if (produtoRepository.existsById(id)) {
            produtoRepository.deleteById(id);
            return true;
        }
        return false;
    }
}
```

5. Camada de Controle (Controller)

Crie o `RestController` para expor as rotas da API e lidar com as requisições HTTP.

5.1. Criando o `ProdutoController`

Crie o arquivo `ProdutoController.java` no pacote `com.exemplo.crudprodutos.controller`.

Java

```
// src/main/java/com/exemplo/crudprodutos/controller/ProdutoController.java
package com.exemplo.crudprodutos.controller;

import com.exemplo.crudprodutos.model.Produto;
import com.exemplo.crudprodutos.service.ProdutoService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/produtos" )
```

```java
public class ProdutoController {

    @Autowired
    private ProdutoService produtoService;

    // 1. POST /produtos - Cadastrar novo produto
    @PostMapping
    public ResponseEntity<Produto> criarProduto(@Valid @RequestBody Produto
produto) {
        Produto novoProduto = produtoService.salvar(produto);
        // Retorna 201 Created (Requisito 5: HTTP Status Code adequado)
        return new ResponseEntity<>(novoProduto, HttpStatus.CREATED);
    }

    // 2. GET /produtos - Listar todos os produtos
    @GetMapping
    public ResponseEntity<List<Produto>> listarProdutos() {
        List<Produto> produtos = produtoService.listarTodos();
        // Retorna 200 OK
        return ResponseEntity.ok(produtos);
    }

    // 3. GET /produtos/{id} - Buscar produto por ID
    @GetMapping("/{id}")
    public ResponseEntity<Produto> buscarProdutoPorId(@PathVariable Long id)
{
        return produtoService.buscarPorId(id)
                // Retorna 200 OK se encontrar, ou 404 Not Found se não
encontrar (Requisito 5)
                .map(ResponseEntity::ok)
                .orElseGet(() -> ResponseEntity.notFound().build());
    }

    // 4. PUT /produtos/{id} - Atualizar produto
    @PutMapping("/{id}")
    public ResponseEntity<Produto> atualizarProduto(@PathVariable Long id,
@Valid @RequestBody Produto produtoDetalhes) {
        Produto produtoAtualizado = produtoService.atualizar(id,
produtoDetalhes);
        if (produtoAtualizado != null) {
            // Retorna 200 OK
            return ResponseEntity.ok(produtoAtualizado);
        }
        // Retorna 404 Not Found se o produto não existir
        return ResponseEntity.notFound().build();
    }

    // 5. DELETE /produtos/{id} - Excluir produto
```

```
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> excluirProduto(@PathVariable Long id) {
        if (produtoService.excluir(id)) {
            // Retorna 204 No Content (Sucesso sem corpo de resposta)
(Requisito 5)
            return ResponseEntity.noContent().build();
        }
        // Retorna 404 Not Found se o produto não existir
        return ResponseEntity.notFound().build();
    }
}
```

6. Configuração e Documentação da API

6.1. Configuração do Banco de Dados H2

Adicione as seguintes linhas ao arquivo `src/main/resources/application.properties` para configurar o H2:

Plain Text

```
# Configuração do H2 Database
spring.datasource.url=jdbc:h2:mem:produtodb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Configuração do JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

6.2. Documentação com Swagger (Springdoc OpenAPI)

A dependência `springdoc-openapi-starter-webmvc-ui` adicionada no passo 1.2 já faz a maior parte do trabalho.

Após iniciar a aplicação, a documentação estará disponível em: `http://localhost:8080/swagger-ui.html`

7. Execução e Teste

7.1. Iniciar a Aplicação

Execute a classe principal `CrudProdutosApplication.java` (ou use o comando `mvn spring-boot:run` no terminal).

7.2. Testando as Rotas

Use o Swagger UI ou uma ferramenta como o Postman para testar as rotas:

Método	Rota	Corpo da Requisição (POST/PUT)	Status Esperado
POST	`/produtos`	`{"nome": "Notebook", "descricao": "Laptop de alta performance", "preco": 5000.00, "quantidadeEstoque": 10}`	`201 Created`
GET	`/produtos`	(Nenhum)	`200 OK`
GET	`/produtos/1`	(Nenhum)	`200 OK` ou `404 Not Found`
PUT	`/produtos/1`	`{"nome": "Notebook Pro", "descricao": "Laptop atualizado", "preco": 5500.00, "quantidadeEstoque": 8}`	`200 OK` ou `404 Not Found`
DELETE	`/produtos/1`	(Nenhum)	`204 No Content` ou `404 Not Found`

8. Finalização e Entrega

8.1. Estrutura do Projeto

Verifique se a estrutura do seu projeto está organizada:

Plain Text

```
crud-produtos/
├── src/main/java/com/exemplo/crudprodutos/
│   ├── CrudProdutosApplication.java
│   ├── controller/
```

```
│   │   └── ProdutoController.java
│   ├── model/
│   │   └── Produto.java
│   ├── repository/
│   │   └── ProdutoRepository.java
│   └── service/
│       └── ProdutoService.java
├── src/main/resources/
│   └── application.properties
└── pom.xml
```

8.2. README e GitHub

Crie um arquivo `README.md` claro no diretório raiz do projeto, contendo:

- Descrição do projeto.
- Tecnologias utilizadas (Spring Boot, Java, H2, Spring Data JPA, Swagger).
- Instruções de como rodar o projeto.
- Lista das rotas da API.

Por fim, inicialize um repositório Git, faça o commit do seu código e envie para o GitHub.

Bash

```bash
# No diretório raiz do projeto
git init
git add .
git commit -m "Implementação inicial do CRUD de Produtos"
git branch -M main
git remote add origin <URL_DO_SEU_REPOSITORIO>
git push -u origin main
```

Este guia cobre todos os requisitos do teste prático, incluindo a organização do código, o uso correto do Spring Boot, a documentação da API, as validações e o uso adequado dos HTTP Status Codes.