Prática 1 - API de Produtos

▼ Objetivo

Nesta prática, vamos desenvolver uma API de gerenciamento de produtos utilizando o framework Spring Boot, implementando as operações básicas de CRUD (Create, Read, Update e Delete).

A API permitirá a criação, consulta, atualização e exclusão de produtos, com cada produto sendo representado por uma entidade contendo atributos como nome, descrição, preço e quantidade.

Endpoints que serão implementados:

GET /produtos

GET /produtos

POST /produtos

PUT /produtos/1

DELETE /produtos/1

▼ 1) Criar projeto Spring Boot

Passo 1:

Para criar o projeto acesse o site: https://start.spring.io/

Passo 2: configure o projeto de acordo com o descrito abaixo.

Project: Maven

Language: Java

Spring Boot: deixe a versão selecionada por padrão.

Group: br.edu.utfpr.nome_aluno

Artifact: api-produto

Package name: deixe o padrão.

Java: no mínimo versão 17.

Passo 3: adicione as dependências

Spring Web

Lombok

Passo 4: gere o arquivo .zip do projeto.

Clique em GENERATE e salve o arquivo no seu computador.

Descompacte o arquivo e use o IntelliJ para abrir a pasta do projeto.

▼ 2) Criando a entidade Produto

Uma classe de modelo no Spring Boot representa uma entidade de dados no domínio da aplicação. Ela define os atributos e comportamentos que descrevem os objetos manipulados no sistema, geralmente mapeando para uma tabela em um banco de dados.

O código abaixo apresenta a classe Produto:

```
package br.edu.utfpr.aluno.api_produto.model;

public class Produto {
    private Long id;
    private String description;
    private Double price;
    private String category;

    // Use a IDE para:
    // Gerar construtor
    // Gerar getters e setters
}
```

Passo 1:

Crie um pacote chamado models dentro do pacote api-produto.

Caminho do pacote: `br.edu.utfpr.aluno.api_produto.models`

Passo 2:

Crie a classe Produto dentro do pacote models e adicione o código acima.

Gere construtor e métodos get e set para a classe Produto.

▼ 3) Controller Produto

Um controller no Spring Boot é uma classe que gerencia as requisições HTTP em uma aplicação web. Ele é responsável por receber essas requisições, processá-las e enviar uma resposta adequada de volta ao cliente.

Um controller é geralmente anotado com `@RestController` ou `@Controller.`

Os métodos dentro dele são mapeados para rotas específicas por meio da anotação `@RequestMapping` ou outras como `@GetMapping` e `@PostMapping`.

Passo 1:

Crie um pacote chamado controllers dentro do pacote api-produto.

Caminho do pacote: `br.edu.utfpr.aluno.api_produto.controllers`

Passo 2:

Crie uma classe chamada ProdutoController dentro do pacote controllers.

Adicione o código abaixo na classe ProdutoController

```
package br.edu.utfpr.aluno.api_produto.controller;

import br.edu.utfpr.aluno.api_produto.model.Produto;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping(path = "/produtos")
public class ProdutoController {
    // Lista de produtos, para simular um banco de dados
    private List<Produto> produtos;

    // Construtor para popular a lista de produtos
    public ProdutoController(){
      this.produtos = new ArrayList<>();
      this.produtos.add(new Produto(1L, "IPHONE 15", 5000.0, "Smartphone"));
      this.produtos.add(new Produto(2L, "Airpods Pro", 2500.0, "Acessórios"));
      this.produtos.add(new Produto(3L, "Notebook i7", 4800.0, "Computadores"));
      this.produtos.add(new Produto(4L, "Cadeira Gamer", 1500.0, "Móveis"));
    }

    // Lista de endpoints

                @GetMapping
    public List<Produto> getAll(){
        return this.produtos;
    }

    //@GetMapping
    //public Produto getOne(Long idProduto){}

    //@PostMapping
    //public void addOne(Produto produto){}

    //@PutMapping
    //public void update(Long idProduto, Produto produto) {}

    //@DeleteMapping
    //public void delete(Long idProduto){}
}
```

Comentários sobre a classe ProdutoController

Foi anotada com @RestController e @RequestMapping

Ela é responsável por tratar requisições ao endpoint /produtos

Processa requisições HTTP do tipo GET, POST, PUT, DELETE para executar operações CRUD sobre a lista de produtos (ainda por implementar)

Endpoint: localhost:8080/produtos

Retorna a lista de produtos no formato JSON


```
[
  {
    "id": 1,
    "description": "IPHONE 15",
    "price": 5000,
    "category": "Smartphone"
  },
  {
    "id": 2,
    "description": "Airpods Pro",
    "price": 2500,
    "category": "Acessórios"
  },
  {
    "id": 3,
    "description": "Notebook i7",
    "price": 4800,
    "category": "Computadores"
  },
  {
    "id": 4,
    "description": "Cadeira Gamer",
    "price": 1500,
    "category": "Móveis"
  }
]
```


Execute a aplicação e acesse o endpoint via navegador e via Postman.

▼ 4) GetMapping (getOne)

Vamos implementar o endpoint para recuperar apenas um produto, por meio de seu identificar id.

Endpoint: localhost:8080/produtos/2

```
@GetMapping(path = "/{id}")
public Produto getOne(@PathVariable(name = "id") Long idProduto){
    Produto produtoEncontrado = this.produtos.stream()
            .filter(produto -> produto.getId().equals(idProduto))
            .findFirst()
            .orElse(null);
    return produtoEncontrado;
}
```

Explicação:

@GetMapping(path = "/{id}"): Esta anotação define que o método responderá a requisições HTTP GET na URL com o formato `/id`, onde `id` é um valor dinâmico.

@PathVariable(name = "id"): O valor dinâmico da URL será capturado e injetado como o argumento `idProduto` no método.

stream() e filter(): O método `stream()` é chamado na lista de produtos para gerar uma sequência de elementos, e `filter()` é usado para filtrar os produtos cujo `id` corresponde ao valor de `idProduto`.

equals(): Como estamos comparando objetos `Long`, utilizamos o método `equals()` para comparar corretamente seus valores.

findFirst(): Retorna o primeiro elemento que corresponde ao filtro, caso exista.

orElse(null): Caso nenhum produto seja encontrado, o método retornará `null`.

▼ 5) PostMapping
Criando um produto novo.

Usar Postman para chamar o endpoint e enviar os dados do novo produto.

```
@PostMapping
public String addOne(@RequestBody Produto produto) {
    if (produto.getDescription() == null || produto.getPrice() < 0){
        return "Descrição ou Preço inválidos";
    } else {
        this.produtos.add(produto);
        return "Produto cadastrado com sucesso";
    }
}
```

Explicação:

1. **@PostMapping**: Esta anotação define que o método responderá a requisições HTTP POST, que geralmente são usadas para criar novos recursos, como um produto, no servidor.

2. **@RequestBody Produto produto**: A anotação `@RequestBody` indica que o objeto `Produto` será construído a partir do corpo da requisição JSON enviada pelo cliente. O Spring Boot faz automaticamente o mapeamento dos dados para o objeto `Produto`.

3. **Validações**: Antes de adicionar o produto à lista, você pode incluir algumas validações, como verificar se o nome do produto é nulo ou se o preço é menor ou igual a zero.

▼ 6) PutMapping

Atualizando dados de um produto existente.

Usar Postman para chamar o endpoint e enviar os dados de alteração do produto.

```
@PutMapping(path="/{id}")
    public void update(@PathVariable(name="id") Long idProduto, @RequestBody
Produto produto) {
        for (Produto p : this.produtos){
            if (p.getId().equals(idProduto)){
                p.setCategory(produto.getCategory());
                p.setDescription(produto.getDescription());
                p.setPrice(produto.getPrice());
                break;
            }
        }
    }
```

Explicação:

1. **@PutMapping(path = "/{id}")**: Adicionada a anotação `@PutMapping` com a definição do caminho para capturar o parâmetro `id` da URL. Isso define o método para responder a uma requisição PUT, usada para atualizar recursos.
2. **@PathVariable Long idProduto**: O `idProduto` vem da URL, então ele deve ser anotado com `@PathVariable`, indicando que o valor é extraído diretamente do caminho da requisição.
3. **Atualização de atributos**: Ao encontrar o produto com o ID correspondente, ele atualiza seus campos com os valores do `produto` recebido na requisição.
4. **`break`**: Após encontrar e atualizar o produto correto, o `break` termina o loop para evitar iterações desnecessárias.

▼ 7) DeletingMapping
Removendo um produto existente.

Usar Postman para chamar o endpoint e enviar o id do produto para remover.

```
@DeleteMapping(path = "/{id}")
public String delete(@PathVariable(name="id") Long idProduto){
    Produto produtoRemover = this.produtos.stream()
            .filter(p->p.getId().equals(idProduto))
            .findFirst()
            .orElse(null);
    if (produtoRemover != null){
        this.produtos.remove(produtoRemover);
        return "Produto removido com sucesso.";
    }
    return "Produto não encontrado";
}
```

Explicação:

1. **@DeleteMapping(path = "/{id}")**: Esta anotação indica que o método será chamado quando uma requisição HTTP `DELETE` for enviada para o endpoint `/produtos/{id}`. O `{id}` é uma variável que será extraída da URL, representando o ID do produto que o cliente deseja remover.

2. **@PathVariable(name="id") Long idProduto**: A anotação `@PathVariable` captura o valor do `id` da URL e o associa à variável `idProduto`. Isso significa que, se o cliente enviar uma requisição para `/produtos/1`, o valor `1` será passado para `idProduto`.

3. **Busca do produto a ser removido**: A lista `this.produtos` é convertida em um stream para realizar uma filtragem.

4. **Verificação se o produto foi encontrado**: A verificação `if (produtoRemover != null)` assegura que o produto foi encontrado na lista. Se `produtoRemover` não for `null`, o produto existe e pode ser removido.

▼ 8) Códigos de Retorno (ResponseEntity)
`ResponseEntity` é uma classe do Spring Framework usada para manipular as respostas HTTP de forma mais detalhada e flexível.

Oferece mais controle em comparação com o retorno direto de objetos simples ou de strings.

Vantagens:

Controle sobre o código de status HTTP: permite retornar códigos de status adequados, como 200 (OK), 201 (Created), 404 (Not Found), etc., proporcionando melhores informações sobre o resultado da operação.

Customização dos cabeçalhos: você pode adicionar informações extras, como localização de um recurso criado ou limites de cache, diretamente nos cabeçalhos da resposta.

Corpo de resposta flexível: você pode retornar qualquer tipo de objeto no corpo da resposta, seja um objeto JSON, uma string ou mesmo um `null`.

Alterando os endpoints para usar ResponseEntity

Abaixo,
`ResponseEntity.ok()` retorna o status 200 (OK) junto com alista de produtos.

```
@GetMapping
public ResponseEntity<List<Produto>> getAll(){
    return ResponseEntity.ok(this.produtos); // 200 OK
}
```

Abaixo, `ResponseEntity.ok()` retorna o status 200 (OK) junto com o produto encontrado.

Se o produto não for encontrado, retorna `ResponseEntity.status(HttpStatus.NOT_FOUND).body(null)`, que envia um código 404.

```
@GetMapping(path = "/{id}")
public ResponseEntity<Produto> getOne(@PathVariable(name = "id") Long idProduto){
        Produto produtoEncontrado = this.produtos.stream()
              .filter(produto -> produto.getId().equals(idProduto))
              .findFirst()
              .orElse(null);

        if (produtoEncontrado == null)
          return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null); // 404
NOT FOUND
        else
          return ResponseEntity.status(HttpStatus.OK).body(produtoEncontrado); //
200 OK
}
```

Abaixo,`HttpStatus.CREATED` para retornar um status 201, que indica que o recurso foi criado com sucesso.

```
@PostMapping
public ResponseEntity<String> addOne(@RequestBody Produto produto) {
    if (produto.getDescription() == null || produto.getPrice() < 0){
                    // 400 Bad Request
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                .body("Descrição ou Preço inválidos");
    } else {
        this.produtos.add(produto);
        // 201 Created
        return ResponseEntity.status(HttpStatus.CREATED)
                .body("Produto cadastrado com sucesso");
    }
}
```

Se o produto for encontrado e atualizado, retorna`ResponseEntity.ok("Produto atualizado com sucesso.")` com status 200.

Caso o produto não seja encontrado, retorna`HttpStatus.NOT_FOUND`.

```
@PutMapping(path="/{id}")
public ResponseEntity<String> update(@PathVariable(name="id") Long idProduto,
@RequestBody Produto produto) {
    for (Produto p : this.produtos){
        if (p.getId().equals(idProduto)){
            p.setDescription(produto.getDescription());
                  p.setQuantity(produto.getQuantity());
            p.setPrice(produto.getPrice());
            p.setCategory(produto.getCategory());
            return ResponseEntity.ok("Produto atualizado com sucesso.");
        }
    }
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Produto não
encontrado.");
}
```

Se o produto for encontrado e removido, retorna`ResponseEntity.ok()` com status 200.

Caso o produto não seja encontrado, `ResponseEntity.status(HttpStatus.NOT_FOUND)` para enviar o status 404.

```
@DeleteMapping(path = "/{id}")
public ResponseEntity<String> delete(@PathVariable(name="id") Long idProduto){
    Produto produtoRemover = this.produtos.stream()
            .filter(p->p.getId().equals(idProduto))
            .findFirst()
            .orElse(null);
    if (produtoRemover != null){
        this.produtos.remove(produtoRemover);
        return ResponseEntity.status(HttpStatus.OK)
                .body("Produto removido com sucesso.");
    }
    return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .body("Produto não encontrado");
}
```

▼ Atividade: Endpoint /produtos/123/venda
Implementar endpoint para realizar uma venda de um item.

```
# POST /produtos/123/venda

# Enviar no corpo da requisição
{
        "quantity": 10
}
```

A venda não será registrada, apenas diminuirá a quantidade em estoque do item.

Necessário validar se há quantidade em estoque para realizar a venda, use os status:

200 OK: venda com sucesso.

400 BAD REQUEST: o usuário enviou uma requisição inválida, pois não há estoque suficiente.