

Construindo um CRUD com Spring Boot e PostgreSQL – Guia Prático para Iniciantes



Autor: Mateus Augusto da Silveira Pinto



RESUMO

Este livro foi criado para quem está começando no desenvolvimento backend com Java e deseja aprender de forma prática como construir uma API RESTful utilizando o framework Spring Boot, além de usar banco de dados PostgreSQL e testar os endpoints criados com o Postman.

Através de um projeto simples aprenderemos a criar um CRUD de controle de estoque. Você vai aprender a estruturar sua aplicação, configurar seu ambiente, lidar com requisições HTTP e persistência de dados.

Escrito de forma didática, com exemplos explicados passo a passo, este guia é ideal para iniciantes que já têm uma base em Java e querem dar o próximo passo no desenvolvimento backend.

Esta primeira versão será feita de forma mais simples, para facilitar o entendimento de iniciantes. No próximo volume haverá tópicos como observabilidade, tratamentos de erros, thymeleaf, documentação com Swagger e mais.

Sumário

1. Apresentação do Projeto
2. Spring Boot
 - 2.1. Dependências
 - 2.1.1. Spring Web
 - 2.1.2. Spring Data JPA
 - 2.1.3. PostgreSQL Driver
 - 2.1.4 Lombok
3. Postman
4. Desenvolvimento do Projeto
 - 4.1. Inicializar
 - 4.2. Camadas do Projeto
 - 4.3. Configuração do Arquivo application.yml
 - 4.4. Criação de um Servidor no pgAdmin 4
 - 4.5. Model
 - 4.6. Repository
 - 4.7. Service
 - 4.8. Controller
 - 4.9. Implementação dos Endpoints
 - 4.9.1. Salvar Produto
 - 4.9.1.1. Repository
 - 4.9.1.2. Service
 - 4.9.1.3. Controller
 - 4.9.1.4. Testando Método “Salvar” no Postman e Verificando no pgAdmin
 - 4.9.2. Listar Todos
 - 4.9.2.1. Service
 - 4.9.3. Buscar por ID
 - 4.9.3.1. Service
 - 4.9.3.2. Controller
 - 4.9.3.3. Testando Método “Buscar por ID” no Postman e Verificando no pgAdmin
 - 4.9.4. Deletar Produto pelo ID
 - 4.9.4.1. Service
 - 4.9.4.2. Controller
 - 4.9.4.3. Testando Método “Deletar Produto pelo ID” no Postman e Verificando no pgAdmin
 - 4.9.5. Atualizar
 - 4.9.5.1. Service
 - 4.9.5.2. Controller
 - 4.9.5.3. Testando Método “Atualizar” no Postman e Verificando no pgAdmin

4.9.6. Buscar por Filtros

4.9.6.1. Repository

4.9.6.2. Service

4.9.6.3. Controller

4.9.6.4. Testando Método “Buscar por Filtros” no Postman e

Verificando no pgAdmin

4.9.7. Remover Quantidade específica de produtos

4.9.7.1. Service

4.9.7.2. Controller

4.9.7.3. Testando Método “Remover Quantidade” no Postman e

Verificando no pgAdmin

5. Código Completo do Projeto

6. Resolução de Alguns Possíveis Problemas Que Podem Aparecer

7. Conclusão

7.1. Próximos passos

Capítulo 1

Apresentação do Projeto

Neste capítulo, será apresentado o projeto a ser desenvolvido ao longo do livro: uma API RESTful para controle de produtos em estoque. Com ele, você aprenderá a estruturar uma aplicação backend usando Spring Boot e PostgreSQL, aplicando boas práticas e conceitos fundamentais de desenvolvimento de APIs.

Neste projeto será permitido realizar as operações básicas de cadastro, consulta, atualização e remoção de produtos, o CRUD (Create, Read, Update, Delete).

Essa API será construída com as seguintes tecnologias: Java 21, Spring Boot e PostgreSQL, além de testar o funcionamento dos endpoints com o Postman.

Nosso projeto permitirá, por exemplo:

- Cadastrar um novo produto, informando nome, descrição, quantidade em estoque e preço.
- Listar todos os produtos disponíveis no sistema.
- Visualizar um produto específico
- Pesquisar por nome e/ou descrição do produto
- Atualizar os dados de um produto existente.
- Deletar um produto do estoque.
- Remover uma quantidade específica do produto existente.

Objetivos com este projeto:

- Aprender a construir uma API REST com Spring Boot de forma prática.
- Persistir dados em um banco relacional utilizando o PostgreSQL.
- Utilizar boas práticas de estruturação de código com Java.
- Criar um projeto funcional que pode servir como base para sistemas maiores.

Capítulo 2

Spring Boot

Spring Boot é um framework Java que facilita a criação de aplicações web modernas e robustas, especialmente APIs REST. Ele faz parte do ecossistema do Spring Framework, mas vem com uma proposta diferente: eliminar a complexidade de configuração.

As aplicações Spring exigiam muita configuração manual (XML, beans, dependências, etc). O Spring Boot resolve isso pois já oferece uma configuração padrão inteligente, para que o desenvolvedor possa focar mais na regra de negócio do sistema e menos em detalhes técnicos de infraestrutura.

Com o Spring Boot você consegue: Criar projetos rapidamente usando o Spring Initializr, rodar a aplicação com um simples `main()` (sem servidor externo, ele embute o Tomcat por padrão), integrar com bancos de dados como PostgreSQL com apenas algumas configurações, usar ferramentas como JPA, segurança, cache e muito mais com facilidade.

O Spring Boot é ideal para: Construção de APIs REST, Aplicações web completas, Microserviços, entre outros.

2.1. Dependências

Dependências são bibliotecas externas que adicionamos para trazer funcionalidades prontas ao nosso sistema. Em vez de programar tudo do zero, usamos essas ferramentas para acelerar o desenvolvimento e seguir boas práticas.

Explicarei a seguir algumas dependências, mais especificamente as dependências que serão utilizadas no projeto abordado neste livro. Serão elas: Spring Web, Spring Data JPA, PostgreSQL Driver e Lombok.

2.1.1 Spring Web

A dependência Spring Web é a responsável por permitir que a gente crie APIs REST ou aplicações web com o Spring Boot. É com ela que conseguimos lidar com requisições HTTP e construir os nossos endpoints. Quando adicionamos `spring-boot-starter-web` ao projeto, o Spring Boot traz junto várias ferramentas importantes, como: Spring MVC, Tomcat embutido e Jackson.

Anotações utilizadas no projeto que fazem parte desta dependência: `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PathVariable`, `@RequestBody`, `@RequestParam`.

Para adicionar a dependência ao projeto, insira o código abaixo no arquivo pom.xml. O pom.xml é o arquivo de configuração usado pelo Maven, onde definimos as dependências, plugins e outras configurações do projeto.

```
<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-web</artifactId>

</dependency>
```

2.1.2. Spring Data JPA

A dependência Spring Data JPA é utilizada para persistir dados em armazenamentos SQL com a API de persistência Java usando Spring Data e Hibernate.

Anotações utilizadas no projeto que fazem parte desta dependência: @Entity, @Table, @Id, @GeneratedValue, @Column, @Repository.

O Spring Data JPA cria automaticamente os comandos SQL com base no seu código Java, mapeia suas classes Java (@Entity) para tabelas no banco e fornece métodos prontos como save(), findById(), findAll() e deleteById() sem precisar escrever uma linha de SQL. Para isso funcionar, na nossa arquitetura iremos primeiro criar a entidade, em seguida o repository, depois o service e por último o controller.

Dentro do arquivo pom.xml esta dependência é inserida da seguinte forma:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2.1.3. PostgreSQL Driver

PostgreSQL Driver é um driver JDBC e R2DBC que permite que programas Java se conectem a um banco de dados PostgreSQL usando código Java padrão e independente de banco de dados.

Para adicionar esta dependência, dentro do arquivo pom.xml insira o código abaixo:

```
<dependency>

  <groupId>org.postgresql</groupId>
```

```
<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>
```

2.1.4. Lombok

Lombok é uma biblioteca de anotações Java que ajuda a reduzir o código repetitivo. No arquivo pom.xml, a dependência é inserida da seguinte forma:

```
<dependency>

  <groupId>org.projectlombok</groupId>

  <artifactId>lombok</artifactId>

  <optional>true</optional>

</dependency>
```

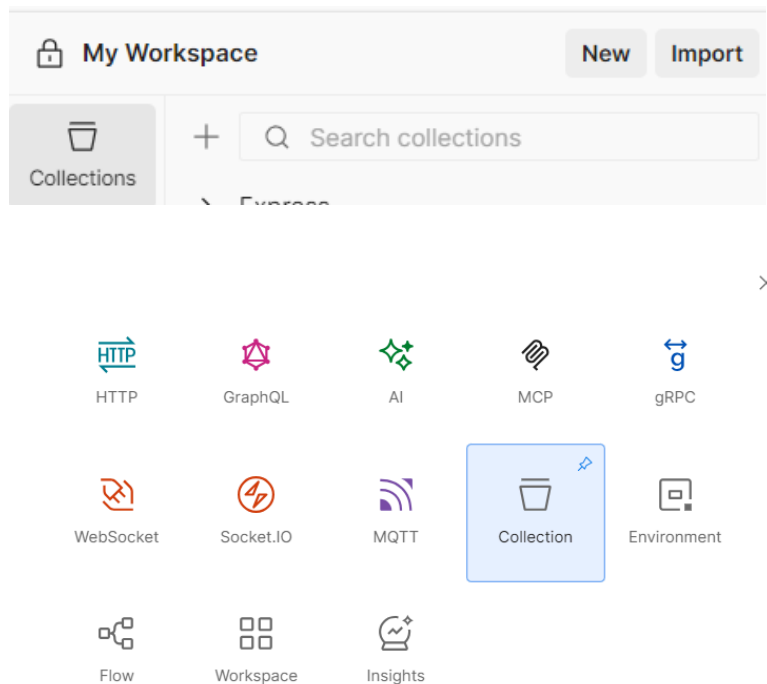
Anotações do Lombok utilizadas no projeto: `@Data`, `@RequiredArgsConstructor`

Capítulo 3

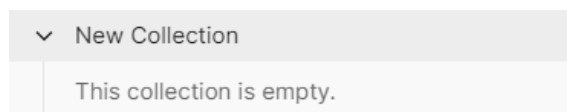
Postman

O Postman é uma ferramenta que permite testar e enviar requisições para APIs de forma simples e prática. Com ele, você pode simular chamadas HTTP (como GET, POST, PUT e DELETE), visualizar as respostas do servidor e verificar se sua API está funcionando corretamente, tudo isso sem precisar criar um front-end. É muito usado por desenvolvedores durante a criação e o teste de APIs REST.

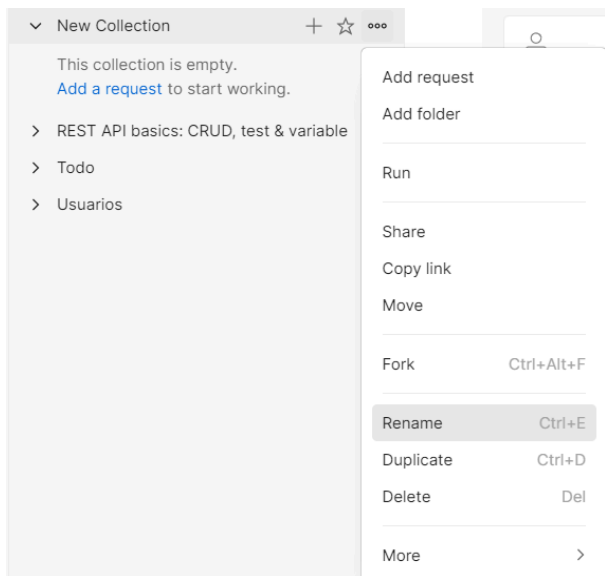
Para iniciar o uso do Postman e criar requests, iremos abrir o app e em seguida criar uma collection. Para isso, basta clicar em New e depois em Collection, como mostrado nas seguintes figuras:



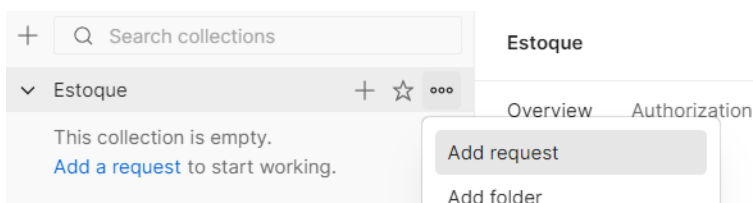
Irá aparecer a nova collection no canto esquerdo da sua tela:



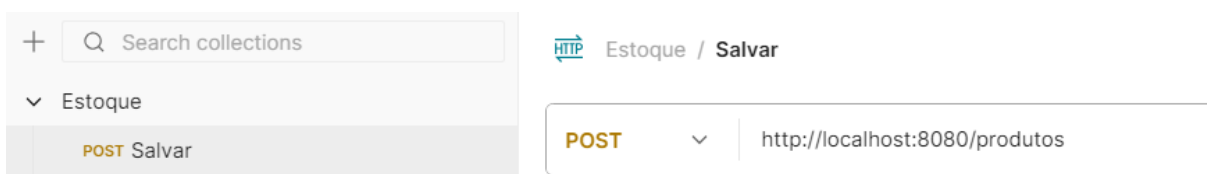
Iremos renomear para “Estoque”:



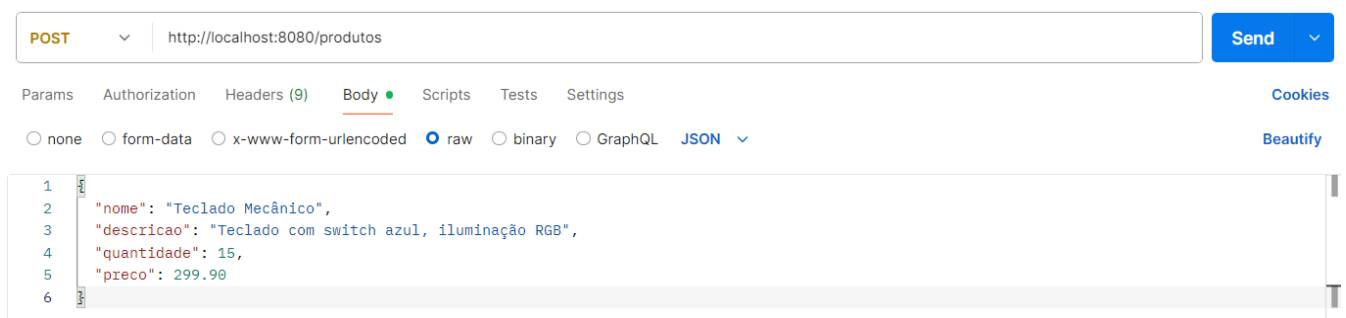
Agora iremos criar nossa primeira request, ela irá representar o endpoint “post” do nosso ProdutoController. Primeiro clicamos nos “3 pontinhos” e Add Request:



Na sequência, configuramos a request da seguinte maneira:



Neste exemplo será criada a request para salvar um produto com http post, por este motivo, em body, colocamos o JSON com os atributos do objeto e clicamos em Send:



Agora será retornado um erro, mas depois que criarmos o método no Spring Boot, receberemos um 201 Created. Esta é apenas uma introdução a criação de requests no Postman, mais a frente veremos como criar cada uma das requests necessárias para este projeto e veremos o retorno deste método funcionando.

Capítulo 4

Desenvolvimento do projeto

4.1. Initializr

O Spring Initializr é uma ferramenta online que ajuda você a criar um projeto Spring Boot do zero de forma rápida e prática. Ele gera uma estrutura básica de projeto com tudo o que você precisa para começar a desenvolver, como:

- Dependências que você escolher (como Spring Web, Spring Data JPA, etc.),
- Arquivos de configuração (como o pom.xml ou build.gradle),
- Estrutura de pastas padrão do Spring Boot.

Você escolhe opções como linguagem (Java, Kotlin ou Groovy), versão do Spring Boot, tipo de projeto (Maven ou Gradle), nome do pacote, e as bibliotecas que quer usar. Depois, ele gera um arquivo .zip com tudo isso pronto para você importar no seu editor (como IntelliJ ou VS Code). Abaixo está a ilustração de como será inicializado o Spring Boot no Initializr:

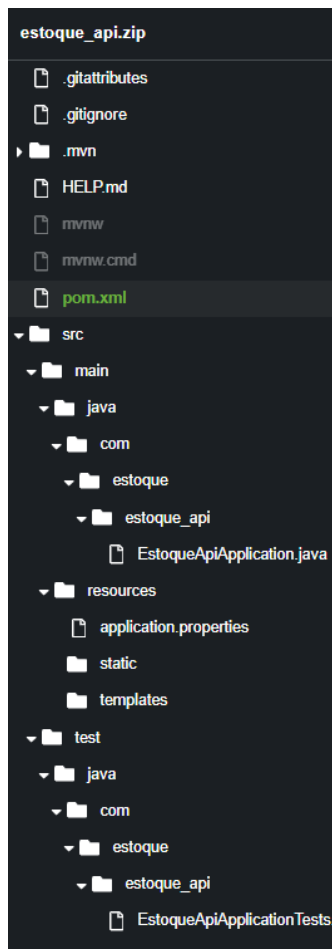
The screenshot shows the Spring Initializr web interface with the following configuration details:

- Project:** ☐ Gradle - Groovy, ☐ Gradle - Kotlin, ☒ Java, ☐ Kotlin, ☐ Groovy
- Build System:** ☒ Maven
- Spring Boot:** ☐ 4.0.0 (SNAPSHOT), ☐ 3.5.4 (SNAPSHOT), ☒ 3.5.3, ☐ 3.4.8 (SNAPSHOT), ☐ 3.4.7
- Project Metadata:**
 - Group: com.estoque
 - Artifact: estoque_api
 - Name: estoque_api
 - Description: Demo project for Spring Boot
 - Package name: com.estoque.estoque_api
- Packaging:** ☒ Jar, ☐ War
- Language:** Java ☐ 24, ☒ 21, ☐ 17
- Dependencies:**
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - PostgreSQL Driver** (SQL): A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
 - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
 - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.

Buttons at the bottom: GENERATE (CTRL + G), EXPLORE (CTRL + SPACE), and a menu icon (three dots).

Após o preenchimento desses campos, basta selecionar o generate e descompactar o arquivo baixado em um local de preferência. Depois de descompactar, abra o projeto em sua IDE de preferência(aqui neste livro usaremos o IntelliJ).

Abaixo temos a estrutura inicial do nosso projeto, ao qual foi gerada automaticamente pelo Spring Initializr:



Faremos a maior parte de nosso projeto dentro de `src/main/java/com/estoque/estoque_api`.

4.2. Camadas do projeto

Nosso projeto será dividido nas seguintes camadas: model, repository, service e controller. Cada camada tem um papel específico, e juntas elas tornam o código mais limpo e organizado. Abaixo a explicação sobre como funcionam:

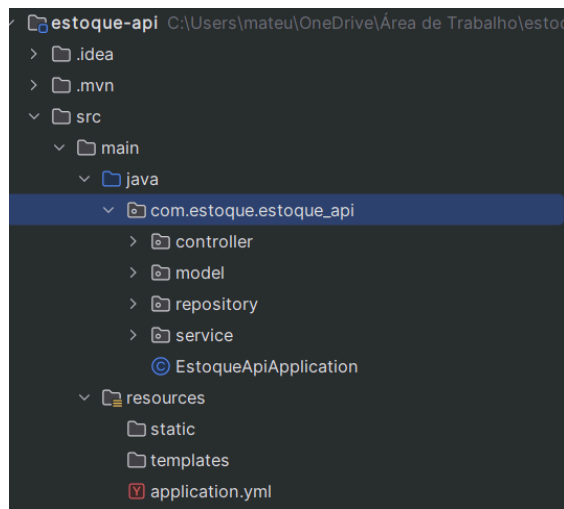
Model: Representa os dados da aplicação, geralmente mapeando as tabelas do banco de dados.

Repository: Responsável pelo acesso ao banco de dados. Utiliza o Spring Data JPA para realizar operações como salvar, buscar, deletar etc.

Service: Onde fica a lógica de negócio da aplicação. Esta camada toma decisões, faz validações e chama o repositório.

Controller: Camada que recebe as requisições HTTP, trata os dados de entrada e envia a resposta. É o ponto de entrada da aplicação.

Abaixo vemos como foi criada a estruturação das pastas no projeto:



4.3. Configuração do arquivo application.yml

Para começar a codificação, primeiro iremos configurar o arquivo pom.xml.

```
spring:

datasource:

  url: jdbc:postgresql://localhost:5433/postgres

  username: postgres

  password: postgres

jpa:

  database-platform: org.hibernate.dialect.PostgreSQLDialect

  hibernate:

    ddl-auto: update

  show-sql: true
```

Na linha “url: jdbc:postgresql://localhost:5433/postgres” utilizei a porta 5433, se você tiver instalado o PostgreSQL em outra porta, você deve utilizar a porta instalada em sua máquina(por padrão é a porta 5432, a menos que tenha alterado, assim como eu).

spring.datasource

Configura a fonte de dados (DataSource), ou seja, os dados para o Spring Boot se conectar ao banco.

- url:
jdbc:postgresql://localhost:5433/postgres
Informa o endereço de conexão:
 - jdbc:postgresql: driver JDBC para PostgreSQL
 - localhost: informa que o banco está rodando localmente
 - 5433: porta onde o PostgreSQL está escutando (a padrão é 5432, só será diferente se você tiver alterado, assim como eu)
 - postgres: nome do banco de dados que está sendo acessado
- username e password:
Credenciais para acessar o banco.

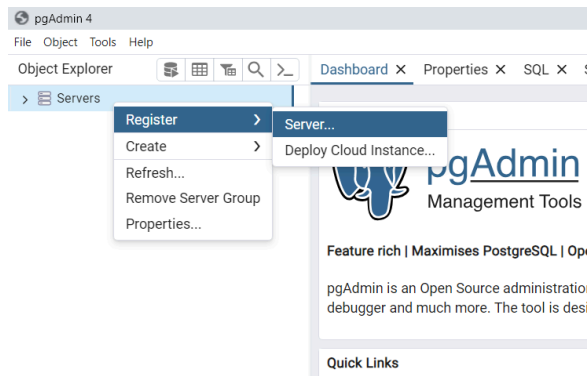
spring.jpa

Configura o comportamento do JPA (Java Persistence API), que é a interface usada para mapear objetos Java para tabelas do banco de dados.

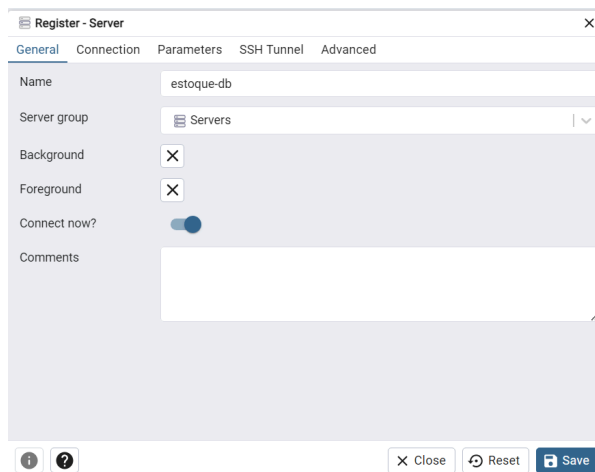
- database-platform:
Define qual dialeto do Hibernate será usado — aqui é o PostgreSQLDialect, que adapta as queries SQL para a sintaxe do PostgreSQL.
- hibernate.ddl-auto: update:
Isso diz ao Hibernate para atualizar automaticamente o esquema do banco conforme as entidades Java. Por exemplo:
 - create: cria o banco toda vez (apaga dados antigos)
 - update: atualiza tabelas e colunas conforme as entidades
 - validate: valida se está tudo certo, sem alterar nada
 - none: não faz nada
- show-sql: true:
Exibe as queries SQL no terminal/logs enquanto a aplicação roda. Muito útil para debug e aprendizado.

4.4. Crie um servidor no PGAdmin 4

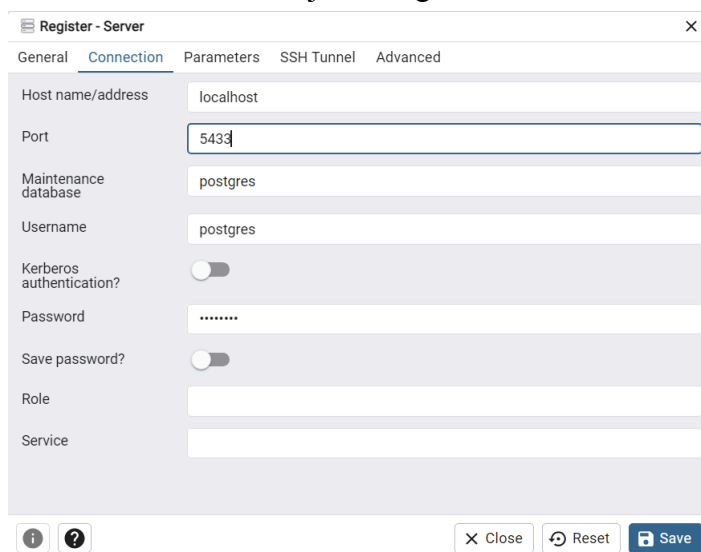
Clique em Register -> Server



Em seguida preencha o campo Name na aba General:



Depois, os campos Port com a porta em que seu banco está rodando, Username e Password com as credenciais desejadas. Agora só salvar:



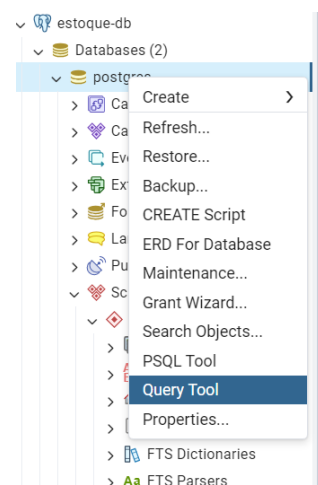
No caso utilizei a porta 5433, pois é a porta que foi definida na instalação do meu PostgreSQL

Para verificar qual porta seu PostgreSQL está ouvindo, basta utilizar o comando “SHOW port;” dentro do psql tool, como mostra na figura abaixo:

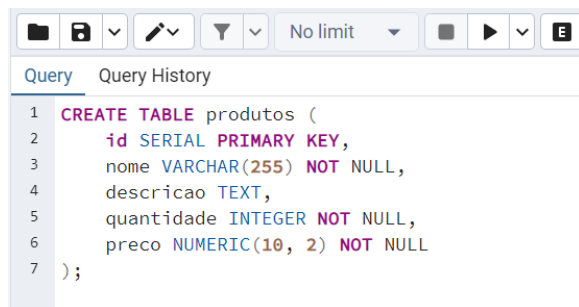
```
postgres=# SHOW port;
 port
-----
 5433
(1 row)

postgres=#
```

Para escrever as queries, clique com o lado direito do mouse em postgres e selecione a opção Query Tool:



Escreva a query abaixo para criar a tabela “produtos” no banco:



Execute o script.

```
Data Output Messages Notifications
CREATE TABLE

Query returned successfully in 467 msec.
```

Tabela criada com sucesso!

Escreva a seguinte query para verificar se a tabela aparece vazia inicialmente:

```
select * from produtos;
```

Use esse comando sempre que quiser verificar se existem produtos na tabela “produtos”.

4.5. Model

A classe Produto foi criada dentro do pacote Model, ela representa a entidade do sistema de controle de estoque. Cada objeto dessa classe corresponde a um registro na tabela de produtos do banco de dados. Essa é uma prática comum em aplicações Spring Boot com Spring Data JPA, onde usamos anotações para mapear os atributos da classe com as colunas da tabela.

```
package com.estoque.estoque_api.model;

import jakarta.persistence.*;
import lombok.Data;
import java.math.BigDecimal;

@Entity
@Table(name = "produtos")
@Data
public class Produto {

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

    @Column(nullable = false)
```

```
private String nome;

private String descricao;

@Column(nullable = false)

private Integer quantidade;

@Column(nullable = false)

private BigDecimal preco;

public void incrementarQuantidade(int quantidadeASomar) {

    if(quantidadeASomar > 0){

        this.quantidade = this.quantidade + quantidadeASomar;

    }

}

}
```

Sobre as anotações presentes neste arquivo:

@Entity: Essa anotação informa ao Spring (via JPA) que a classe Produto é uma entidade, ou seja, que ela representa uma tabela no banco de dados.

@Table(name = "produtos"): Define o nome da tabela no banco de dados como produtos. Se essa anotação fosse omitida, o nome da tabela seria produto (o nome da classe em minúsculo, por padrão).

@Data (Lombok): Gera automaticamente Getters e Setters para todos os campos, além de métodos equals(), hashCode() e toString(). Isso deixa o código mais limpo e evita repetições desnecessárias.

@Id: Marca o campo id como chave primária da tabela.

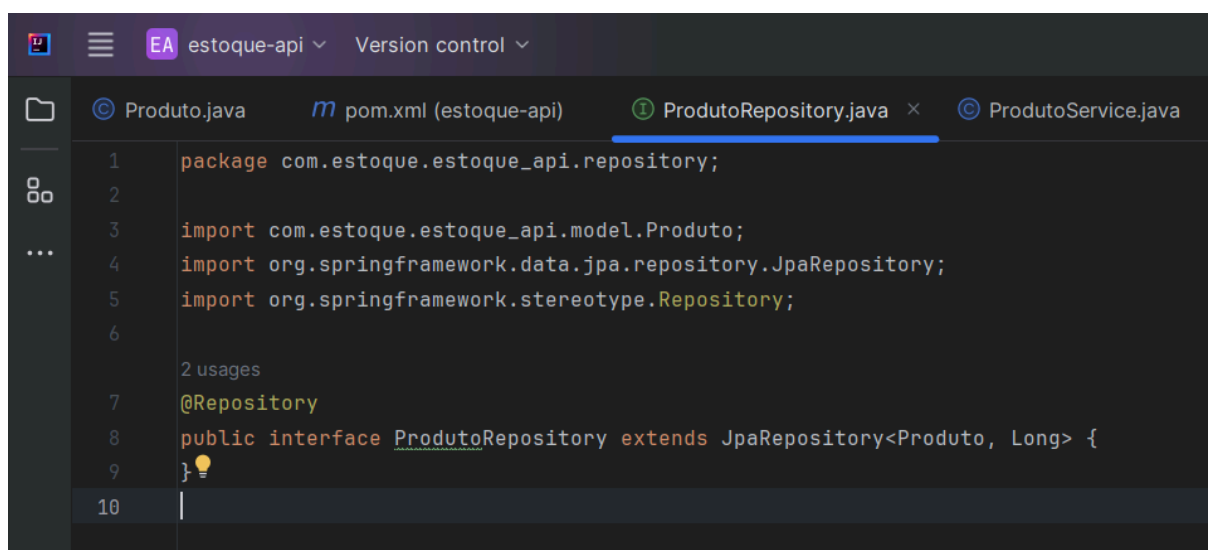
@GeneratedValue(strategy = GenerationType.IDENTITY): Indica que o valor do id será gerado automaticamente pelo banco de dados (estratégia de auto incremento).

`@Column(nullable = false)`: Define que os campos nome, quantidade e preco não podem ser nulos no banco de dados. Ou seja, esses dados são obrigatórios ao cadastrar um novo produto.

O método `incrementarQuantidade`: serve para aumentar a quantidade em estoque de um produto. Ele recebe um valor inteiro que representa quanto queremos somar à quantidade atual. Mais a frente iremos entender o porque este método foi criado.

4.6. Repository

A interface `ProdutoRepository` é responsável por realizar a comunicação com o banco de dados, utilizando o Spring Data JPA. Ela representa a camada de repositório da aplicação, que é responsável por salvar, buscar, atualizar e deletar dados da entidade `Produto`.



```
1 package com.estoque.estoque_api.repository;
2
3 import com.estoque.estoque_api.model.Produto;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 @Repository
8 public interface ProdutoRepository extends JpaRepository<Produto, Long> {
9 }
10
```

`JpaRepository` é uma interface do Spring Data JPA que já fornece todos os métodos básicos de um CRUD, como: `save()`, `findById()`, `findAll()`, `deleteById()`, entre outros.

Os parâmetros `<Produto, Long>` indicam que: A entidade gerenciada será `Produto` e o tipo da chave primária (`id`) é `Long`.

`@Repository`: Acesso a dados e integração com o banco. Essa anotação marca a interface como um componente de persistência, permitindo que o Spring faça a injeção automática dela onde for necessário (por exemplo, na camada service).

Obs.: mesmo que não colocasse essa anotação, o Spring Boot reconheceria a interface automaticamente por causa do `JpaRepository`, mas é uma boa prática usá-la.

4.7. Service

A classe `ProdutoService` será iniciada da seguinte forma:

```

package com.estoque.estoque_api.service;

import com.estoque.estoque_api.model.Produto;
import com.estoque.estoque_api.repository.ProdutoRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class ProdutoService {
    private final ProdutoRepository repository;
}

```

As annotations utilizadas serão: `@Service` e `@RequiredArgsConstructor`.

`@Service`: é usada para marcar uma classe de serviço, que faz parte da camada de negócio da aplicação.

`@RequiredArgsConstructor`: A anotação `@RequiredArgsConstructor` vem do Lombok e serve para gerar automaticamente um construtor com os atributos final (ou `@NonNull`) da classe. Sendo assim, nosso `ProdutoService` terá um construtor de forma automática com o parâmetro e atributo “repository”, como mostrado abaixo:

```

public ProdutoService(ProdutoRepository repository) {
    this.repository = repository;
}

```

4.8. Controller

A classe `ProdutoController` será iniciada da seguinte forma:

```

package com.estoque.estoque_api.controller;

import com.estoque.estoque_api.model.Produto;
import com.estoque.estoque_api.service.ProdutoService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.util.List;
import java.util.Optional;

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

```

```
private final ProdutoService service;  
}
```

Algumas das annotations utilizadas serão: `@RestController` e `@RequestMapping`.

`@RestController`: Indica que a classe trata requisições HTTP e retorna dados no corpo da resposta (geralmente JSON)

`@RequestMapping("/produtos")`: Define a rota base para todos os métodos da classe. Ou seja, todos os endpoints da classe começarão com `"/produtos"`.

4.9. Implementação dos Endpoints

Neste subcapítulo veremos como foram criados os métodos deste projeto

4.9.1. Salvar Produto

4.9.1.1. Repository

Vamos precisar criar um método no repository para fazer a busca por Nome e Descrição dentro do service, pois quando vamos salvar um novo produto, queremos verificar se já existe um produto com o mesmo nome e descrição no banco de dados. Se existir, não criamos um novo registro, apenas atualizamos a quantidade do produto existente.

```
@Repository  
public interface ProdutoRepository extends JpaRepository<Produto, Long> {  
    Optional<Produto> findByNomeAndDescricao(String nome, String descricao);  
}
```

Para isso, precisamos de uma forma de buscar um produto pelo nome e pela descrição ao mesmo tempo. E aí entra o método `findByNomeAndDescricao`. No Spring Data JPA, não precisamos escrever consultas SQL manualmente para buscas simples. O Spring consegue gerar automaticamente a consulta com base no nome do método.

4.9.1.2. Service

O método salvar, dentro do service, é responsável por salvar um produto no banco de dados, e com uma lógica para evitar a duplicação de produtos com o mesmo nome e descrição.

```
public Produto salvar(Produto produto){  
    Optional<Produto> existente = repository.findByNomeAndDescricao(produto.getNome(),  
produto.getDescricao());  
  
    if (existente.isPresent()){  
        Produto produtoExistente = existente.get();  
        produtoExistente.incrementarQuantidade(produto.getQuantidade());  
        return repository.save(produtoExistente);  
    }  
}
```

```

    } else {
        return repository.save(produto);
    }
}

```

O método começa buscando no banco de dados se já existe um produto com o mesmo nome e descrição que o produto que queremos salvar. Essa busca é feita através do repositório, que retorna um `Optional<Produto>`. Esse `Optional` pode conter o produto encontrado, caso exista, ou estar vazio se nenhum produto corresponder aos critérios.

Em seguida, o método verifica se o produto já existe. Se o `Optional` contiver um produto, significa que já há um registro no banco com as mesmas características. Nesse caso, o método recupera o produto existente, chama o método `incrementarQuantidade` para somar a quantidade do novo produto à quantidade já armazenada e, por fim, salva essa atualização no banco, retornando o produto atualizado. Este é o motivo de termos criado o método `incrementarQuantidade` na classe `Produto`, assim a lógica de alteração da quantidade fica dentro da própria classe (princípio do Encapsulamento), evitando espalhar a mesma lógica em várias partes da aplicação. Por outro lado, se não existir nenhum produto com o mesmo nome e descrição, o método simplesmente salva o produto recebido como novo registro no banco e o retorna.

4.9.1.3. Controller

O método “salvar”, dentro da classe `ProdutoController`, é responsável por receber uma requisição HTTP POST para cadastrar um novo produto no sistema.

```

@PostMapping
public ResponseEntity<Produto> salvar(@RequestBody Produto produto) {
    Produto salvo = service.salvar(produto);
    URI location = URI.create("/produtos/" + salvo.getId());
    return ResponseEntity
        .created(location)
        .body(salvo);
}

```

Quando o cliente envia os dados do produto no corpo da requisição (normalmente em formato JSON), o Spring automaticamente converte esses dados para um objeto `Produto`. Essa conversão automática acontece por causa de um recurso do Spring chamado Jackson, que é uma biblioteca de desserialização (o JSON recebido transformado em um objeto Java (`Produto`)) e serialização (um objeto Java transformado em JSON para a resposta) de JSON.

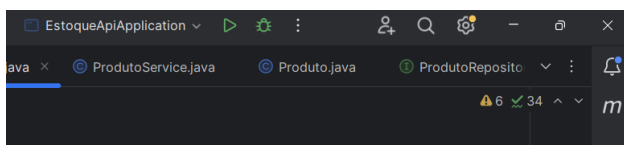
Em seguida, o método chama o serviço responsável por salvar o produto. Esse serviço pode criar um novo registro no banco de dados ou, caso o produto já exista (com o mesmo nome e descrição), ele simplesmente atualiza a quantidade em estoque.

Após salvar o produto, o método cria um endereço (URI) que representa o local onde o novo produto foi armazenado, usando o ID gerado pelo banco. Esse endereço é incluído no cabeçalho da resposta para informar ao cliente onde o recurso pode ser encontrado.

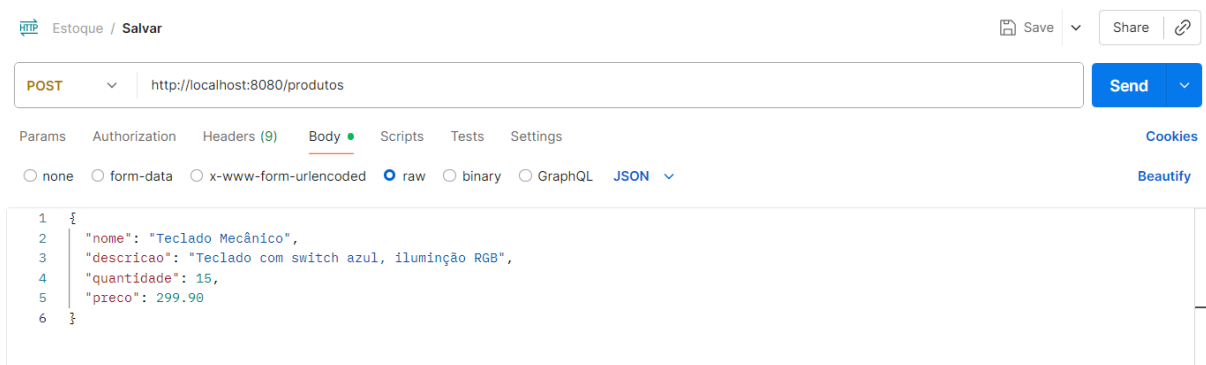
Por fim, o método retorna uma resposta HTTP com o status **201 Created**, indicando que o cadastro foi realizado com sucesso, e inclui no corpo da resposta os dados completos do produto que acabou de ser salvo.

4.9.1.4. Testando Método “Salvar” no Postman e verificando no pgAdmin

Este é o método que criamos no capítulo 3 deste livro, para ver se está funcionando, iremos primeiramente apertar em run e ver se o código compila:



Após isso, vamos no Postman e clicamos em send:



E aqui está o resultado obtido, um “201 Created” como esperado:



O “Created” foi o retorno obtido porque pedimos que fosse retornado no código do ProdutoController.

```
return ResponseEntity
    .created(location)
    .body(salvo);
```


Além disso, retornar a location é uma boa prática RESTful, pois facilita para o cliente saber onde buscar, atualizar ou deletar o recurso depois.

Body	Cookies	Headers (6)	Test Results	⌚	201 Created
Key			Value		
Location			/produtos/1		

Ao verificar o banco de dados com o comando “select * from produtos;”, vemos que o produto foi adicionado na tabela de produtos:

Query

Query History

1 select * from produtos;

Data Output

Messages

Notifications

4.9.2. Listar Todos

Este método será usado para listar todos produtos, ele será útil dentro do método de buscarFiltro no ProdutoController, ao qual lista todos os produtos existentes no banco caso não passe nenhum produto no parâmetro.

4.9.2.1. Service

Dentro de ProdutoService escrevemos o seguinte código:

```
public List<Produto> listarTodos(){  
    return repository.findAll();  
}
```

Para isso, ele utiliza o repositório, que é responsável por acessar os dados diretamente. O método `findAll()` do repositório retorna todos os registros da tabela de produtos, e essa lista é entregue para quem chamou o método, permitindo que seja exibida ou processada conforme necessário. Mais a frente veremos o uso dele dentro do método `buscarPorFiltro` da classe `ProdutoController`.

4.9.3. Buscar por ID

O método `buscarPorId` serve para encontrar um produto específico no banco de dados a partir do seu identificador único (ID).

4.9.3.1. Service

Dentro da classe `ProdutoService`, ele recebe esse ID como parâmetro e utiliza o repositório para buscar o produto correspondente.

```
public Optional<Produto> buscarPorId(Long id){  
    return repository.findById(id);  
}
```

Como nem sempre o produto pode existir, o método retorna um `Optional<Produto>`. Isso significa que o resultado pode conter o produto encontrado ou estar vazio, caso nenhum produto com aquele ID seja localizado. Dessa forma, o código que chama esse método pode verificar se o produto foi realmente encontrado antes de tentar usá-lo.

4.9.3.2. Controller

O método “`buscarPorId`”, dentro de `ProdutoController`, responde a requisições HTTP do tipo GET feitas ao endpoint `/produtos/{id}`, onde `{id}` é um valor variável que representa o identificador do produto que se deseja buscar.

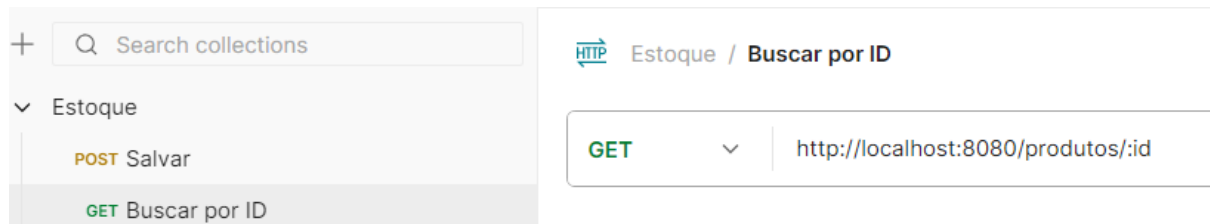
```
@GetMapping("/{id}")  
public ResponseEntity<Produto> buscarPorId(@PathVariable Long id) {  
    return service.buscarPorId(id)  
        .map(ResponseEntity::ok)  
        .orElse(ResponseEntity.notFound().build());  
}
```

Quando o método é chamado, ele solicita ao serviço que procure o produto correspondente ao ID informado. O resultado vem encapsulado em um `Optional`.

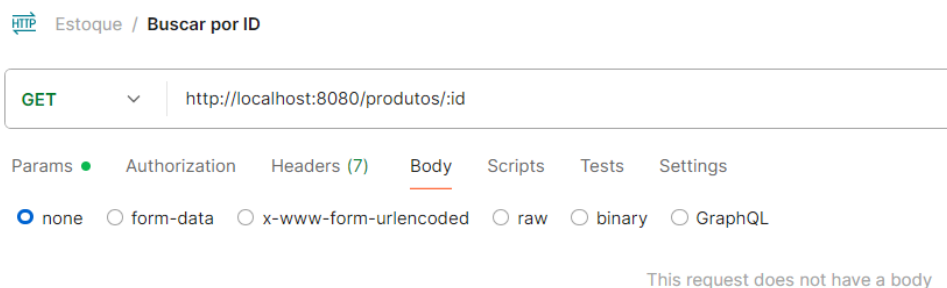
Se o produto for encontrado, o método retorna uma resposta HTTP com o status 200 OK junto com os dados do produto no corpo da resposta. Caso contrário, se o produto não existir, a resposta será um status 404 Not Found, indicando que o recurso solicitado não foi encontrado.

4.9.3.3. Testando Método “Buscar por ID” no Postman e verificando no pgAdmin

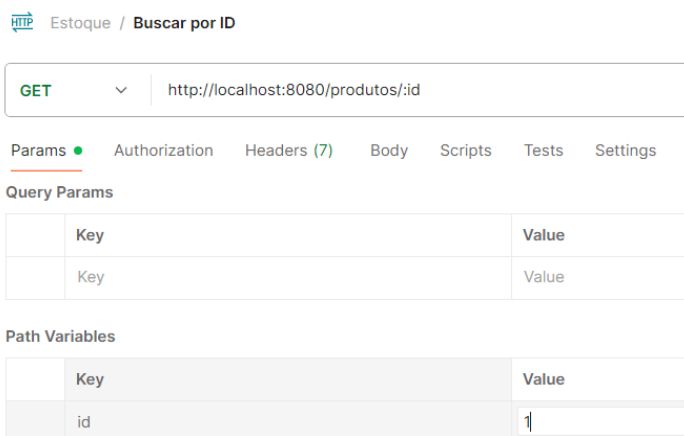
O método Buscar por ID foi criado da seguinte maneira:



Neste método, o body fica vazio, já que o método GET é usado apenas para buscar informações, e não para enviar dados no corpo da requisição.:



Para escolher qual id queremos encontrar, é necessário preencher o campo Value de Path Variables. Neste primeiro caso iremos buscar pelo ID “1” como a seguir:



Se o produto for encontrado, será retornado um status “200 OK” com o JSON do objeto buscado:



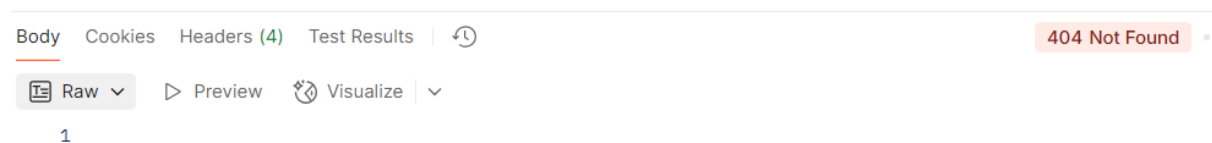
Esta resposta conseguimos no retorno do método no ProdutoController:

```
return service.buscarPorId(id).optional().orElse(ResponseEntity.notFound().build());
```

Se tentarmos buscar pelo ID “2” neste momento, obteremos um retorno “404 Not Found”:

Path Variables

	Key	Value	Description
	id	2	Description



Isso acontece porque neste momento ainda não temos um segundo produto salvo no banco.

Salvaremos mais um produto no banco para que possamos realizar mais testes. Vamos salvar um mouse com o seguinte body:

HTTP Estoque / Salvar

POST http://localhost:8080/produtos

Send

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "nome": "Mouse",
3   "descricao": "Mouse preto",
4   "quantidade": 5,
5   "preco": 100.00
6 }
```

Body Cookies Headers (6) Test Results

201 Created • 276 ms • 271 B

JSON Preview Visualize

```
1 {
2   "id": 2,
3   "nome": "Mouse",
4   "descricao": "Mouse preto",
5   "quantidade": 5,
6   "preco": 100.00
7 }
```

Agora quando tentamos buscar pelo ID “2” conseguimos uma resposta diferente, um “200 OK”:

HTTP Estoque / Buscar por ID

GET http://localhost:8080/produtos/id

Send

Params Authorization Headers (7) Body Scripts Tests Settings

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Path Variables

	Key	Value	Description	...	Bulk Edit
	id	2	Description		

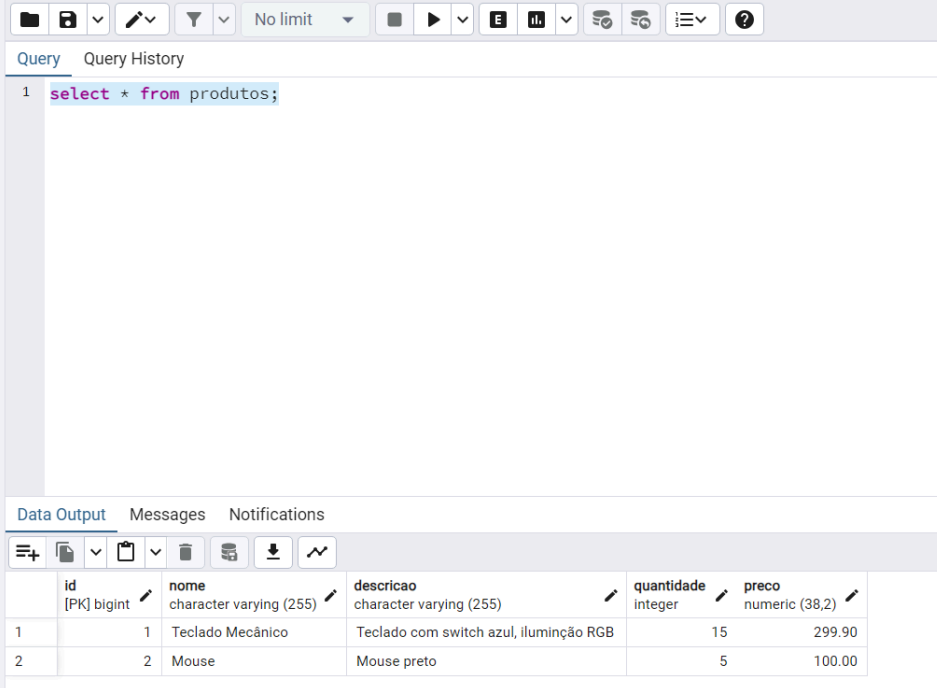
Body Cookies Headers (5) Test Results

200 OK • 18 ms • 243 B

JSON Preview Visualize

```
1 {
2   "id": 2,
3   "nome": "Mouse",
4   "descricao": "Mouse preto",
5   "quantidade": 5,
6   "preco": 100.00
7 }
```

Se olharmos no banco, vemos que os resultados estão sendo obtidos de forma correta:



The screenshot shows a database query interface. At the top, there's a toolbar with various icons. Below it, a 'Query' tab is active, displaying the SQL query: `select * from produtos;`. Below the query, there's a 'Data Output' tab showing the results of the query in a table format. The table has five columns: `id` (PK bigint), `nome` (character varying (255)), `descricao` (character varying (255)), `quantidade` (integer), and `preco` (numeric (38,2)). There are two rows of data: Row 1 with `id` 1, `nome` 'Teclado Mecânico', `descricao` 'Teclado com switch azul, iluminação RGB', `quantidade` 15, and `preco` 299.90; Row 2 with `id` 2, `nome` 'Mouse', `descricao` 'Mouse preto', `quantidade` 5, and `preco` 100.00.

	id [PK] bigint	nome character varying (255)	descricao character varying (255)	quantidade integer	preco numeric (38,2)
1	1	Teclado Mecânico	Teclado com switch azul, iluminação RGB	15	299.90
2	2	Mouse	Mouse preto	5	100.00

Teclado Mecânico com ID 1 e Mouse com ID 2.

4.9.4. Deletar Produto pelo ID

O método “deletar” tem a função de remover um produto do banco de dados com base no seu identificador (ID).

4.9.4.1. Service

Dentro da classe `ProdutoService`, ele recebe esse ID como parâmetro e utiliza o repositório para executar a exclusão do registro correspondente.

```
public void deletar(Long id){
    repository.deleteById(id);
}
```

Ao chamar esse método, o produto com o ID informado será deletado, se existir. Caso contrário, o repositório não fará nenhuma alteração.

4.9.4.2. Controller

Dentro da classe `ProdutoController`, o método “deletar”, responde a requisições HTTP do tipo DELETE feitas ao endpoint `/produtos/{id}`, onde `{id}` é o identificador do produto que se deseja remover.

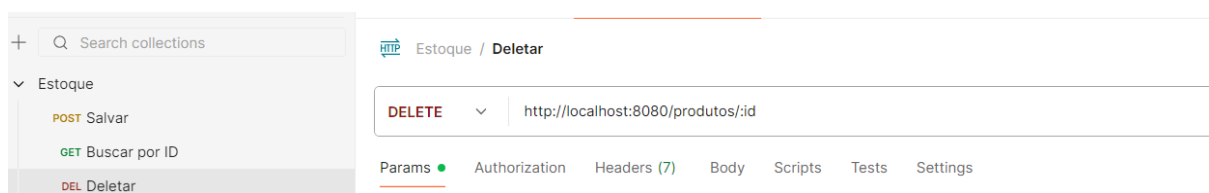
```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deletar(@PathVariable Long id){
    service.deletar(id);
    return ResponseEntity.noContent().build();
}
```

```
}
```

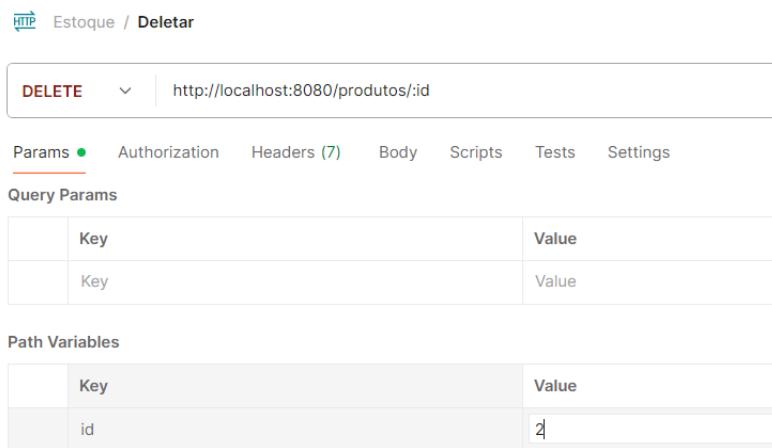
Quando chamado, ele solicita ao serviço que delete o produto com o ID informado. Após a exclusão, o método retorna uma resposta HTTP com o status 204 No Content, indicando que a operação foi realizada com sucesso, mas que não há conteúdo para ser retornado na resposta.

4.9.4.3. Testando Método “Deletar Produto pelo ID” no Postman e verificando no pgAdmin

No Postman, criaremos o método para deletar com o nome = “Deletar”, method = “DELETE” e url = “http://localhost:8080/produtos/:id”, como mostrado na imagem:



Em Path Variables colocamos id e em Value o valor do id que queremos passar por parâmetro no método que criamos em ProdutoController:



Iremos deletar o produto de id = 2. O retorno de status que será obtido é um “204 No Content”:

Estoque / Deletar

DELETE ▼ http://localhost:8080/produtos/id

Params ● Authorization Headers (7) Body Scripts Tests Settings

Query Params

	Key	Value	Descr
	Key	Value	Descr

Path Variables

	Key	Value	Descr
	id	2	Descr

Body Cookies Headers (3) Test Results ↺ 204 No Content

Raw ▼ ▶ Preview 🔄 Visualize ▼

Se olharmos no pgAdmin, veremos que o produto foi excluído com sucesso(caso esteja correto):

Query Query History

1 `select * from produtos;`

Data Output Messages Notifications

	id [PK] bigint	nome character varying (255)	descricao character varying (255)	quantidade integer	preco numeric (38,2)
1	1	Teclado Mecânico	Teclado com switch azul, iluminação RGB	15	299.90

Note que o Mouse não está mais presente na lista de produtos.

Outro exemplo seria se o produto não existisse na tabela. Nesse caso, o retorno seria o mesmo “204 No Content”, já que foi a abordagem que escolhemos ao construir nosso projeto.

4.9.5. Atualizar

O método atualizar é responsável por modificar os dados de um produto já existente no banco de dados.

4.9.5.1. Service

Este método, dentro da camada Service, recebe o identificador do produto que será atualizado e um objeto com os novos dados.

```
public Optional<Produto> atualizar(Long id, Produto produtoAtualizado){  
    return repository.findById(id).map(produtoExistente -> {
```



```

    produtoExistente.setNome(produtoAtualizado.getNome());
    produtoExistente.setDescricao(produtoAtualizado.getDescricao());
    produtoExistente.setQuantidade(produtoAtualizado.getQuantidade());
    produtoExistente.setPreco(produtoAtualizado.getPreco());
    return repository.save(produtoExistente);
});
}

```

Primeiro, o método tenta encontrar o produto pelo ID. Caso ele exista, o método atualiza os campos nome, descricao, quantidade e preco com os valores fornecidos no objeto de atualização. Depois disso, salva as alterações no banco e retorna o produto atualizado.

Se o produto com o ID informado não for encontrado, o método retorna um Optional vazio, indicando que a atualização não pôde ser realizada porque o produto não existe.

4.9.5.2. Controller

Na camada Controller, o método “atualizar” atende às requisições HTTP do tipo PUT no endpoint /produtos/{id}, onde {id} representa o identificador do produto que será atualizado.

```

@PutMapping("/{id}")
public ResponseEntity<Produto> atualizar(@PathVariable Long id, @RequestBody Produto produto){
    return service.atualizar(id, produto)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}

```

Quando chamado, ele recebe no corpo da requisição os novos dados do produto, que são convertidos automaticamente para um objeto Produto. Em seguida, ele solicita ao serviço que faça a atualização do produto com o ID especificado.

Se o produto for encontrado e atualizado com sucesso, o método responde com o status HTTP 200 OK, enviando o produto atualizado no corpo da resposta. Caso o produto não exista, a resposta será um status 404 Not Found, indicando que a atualização não pôde ser realizada porque o recurso não foi encontrado.

4.9.5.3. Testando Método “Atualizar” no Postman e verificando no pgAdmin

Para criar o método atualizar iremos utilizar o método HTTP PUT, com nome = “Atualizar” e a URL = “http://localhost:8080/produtos/:id”. Neste método, precisaremos passar o Id do produto que queremos modificar e atualizar os valores desejados no body.

Nosso banco só tem o produto de Id = “1” no momento, sendo assim, passaremos este valor no parâmetro:

HTTP Estoque / Atualizar

PUT ▼ http://localhost:8080/produtos/:id

Params ● Authorization Headers (9) Body ● Scripts Tests Settings

Query Params

	Key	Value
	Key	Value

Path Variables

	Key	Value
	id	1

No Body selecionamos raw e JSON, assim como no método de “salvar”:

+ Search collections

Estoque

POST Salvar
GET Buscar por ID
DEL Deletar
PUT Atualizar

HTTP Estoque / Atualizar

PUT ▼ http://localhost:8080/produtos/:id

Params ● Authorization Headers (9) Body ● Scripts Tests Settings

☐ none
☐ form-data
☐ x-www-form-urlencoded
☒ raw
☐ binary
☐ GraphQL
JSON ▼

Agora escreveremos o body de acordo com a imagem:

HTTP Estoque / Atualizar

PUT ▼ http://localhost:8080/produtos/:id

Params ● Authorization Headers (9) Body ● Scripts Tests Settings

☐ none
☐ form-data
☐ x-www-form-urlencoded
☒ raw
☐ binary
☐ GraphQL
JSON ▼

```

1  {
2      "nome": "Mouse",
3      "descricao": "Mouse com iluminação RGB",
4      "quantidade": 10,
5      "preco": 100.00
6  }
```

Ao clicar em Send, obtemos o retorno “200 OK” com o produto atualizado:

```
Body Cookies Headers (5) Test Results 200 OK
{ } JSON Preview Visualize
1 {
2   "id": 1,
3   "nome": "Teclado Mecânico",
4   "descricao": "Teclado sem iluminação RGB",
5   "quantidade": 10,
6   "preco": 110.00
7 }
```

No banco, antes de atualizar tínhamos o produto de Id = 1 com esses valores:

Data Output Messages Notifications					
	id [PK] bigint	nome character varying (255)	descricao character varying (255)	quantidade integer	preco numeric (38,2)
1	1	Teclado Mecânico	Teclado com switch azul, iluminação RGB	15	299.90

Após atualizar, temos novos valores:

Data Output Messages Notifications					
	id [PK] bigint	nome character varying (255)	descricao character varying (255)	quantidade integer	preco numeric (38,2)
1	1	Teclado Mecânico	Teclado sem iluminação RGB	10	110.00

4.9.6. Buscar por Filtros

Nosso próximo passo será realizar a busca por filtros, este recurso permite que o usuário consulte os produtos de maneira dinâmica, podendo informar filtros opcionais de nome e/ou descrição. Quando nenhum filtro é enviado, todos os produtos cadastrados são retornados.

4.9.6.1. Repository

O método buscarPorFiltros, na camada Repository, realiza uma busca flexível no banco de dados de produtos, permitindo que o usuário filtre por nome, ou filtre por descrição, ou filtre por ambos, ou não envie nenhum filtro e receba todos os produtos.

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {

    @Query("SELECT p FROM Produto p " +
        "WHERE (:nome IS NULL OR LOWER(COALESCE(p.nome, '')) LIKE LOWER(CONCAT('%', :nome, '%'))) " +
```

```

"AND (:descricao IS NULL OR LOWER(COALESCE(p.descricao, '')) LIKE LOWER(CONCAT('%', :descricao, '%')))"
List<Produto> buscarPorFiltros(@Param("nome") String nome, @Param("descricao") String descricao);
}

```

O `@Query(...)` diz ao Spring que vamos usar uma consulta personalizada em JPQL (Java Persistence Query Language), e não aquelas automáticas como `findByNomeAndDescricao`.

Sobre a query: A consulta `SELECT p FROM Produto p` seleciona todos os objetos do tipo `Produto` da tabela, representados pela letra `p`. A cláusula `WHERE` define os critérios de filtragem com base nos parâmetros `nome` e `descricao` fornecidos na requisição. O trecho que diz `(:nome IS NULL OR LOWER(COALESCE(p.nome, '')) LIKE LOWER(CONCAT('%', :nome, '%')))` significa que, se o parâmetro `nome` não for informado (ou seja, for `null`), o filtro é ignorado. Caso contrário, a consulta busca produtos cujo nome contenha o texto informado, sem diferenciar letras maiúsculas de minúsculas. A função `COALESCE(p.nome, '')` é usada para evitar erros caso algum produto tenha o campo `nome` com valor `null` no banco de dados, substituindo esse valor por uma string vazia apenas durante a comparação. Por exemplo, se o parâmetro `nome` for "limao", serão retornados resultados como "Limaó Azedo", "limao capeta" e outros semelhantes. A parte `AND (:descricao IS NULL OR LOWER(COALESCE(p.descricao, '')) LIKE LOWER(CONCAT('%', :descricao, '%')))` segue a mesma lógica anterior, aplicando o filtro de forma semelhante ao campo `descricao`.

4.9.6.2. Service

O método `buscarPorFiltros` permite pesquisar produtos no banco de dados com base em filtros opcionais de nome e descrição. Antes de realizar a busca, o método verifica se os valores de nome e descrição foram informados.

```

public List<Produto> buscarPorFiltros(String nome, String descricao) {
    if (nome != null) {
        nome = nome.toLowerCase();
    }
    if (descricao != null) {
        descricao = descricao.toLowerCase();
    }
    return repository.buscarPorFiltros(nome, descricao);
}

```

Se algum desses valores não for nulo, ele converte o texto para letras minúsculas para garantir que a busca seja feita de forma case-insensitive, ou seja, sem diferenciar maiúsculas de minúsculas. Depois, o método chama o repositório para executar a consulta personalizada, que retorna uma lista de produtos que correspondem aos filtros informados. Caso os filtros sejam nulos, a consulta retorna todos os produtos. Sendo assim, o método oferece uma forma flexível e eficiente de buscar produtos conforme critérios fornecidos pelo usuário.

4.9.6.3. Controller

Antes de criar este método no ProdutoController, iremos apagar o código do método listarTodos que está dentro da classe ProdutoController. Atenção: apagar apenas dentro do arquivo ProdutoController. Iremos apagar pois agora juntaremos o listarTodos com buscarPorFiltros, já que este novo método também terá a capacidade de listarTodos.

Este método responde a requisições HTTP GET feitas para o endpoint /produtos:

```
@GetMapping
public ResponseEntity<List<Produto>> buscarPorFiltros(@RequestParam(required = false) String nome,
                                                    @RequestParam(required = false) String descricao){

    List<Produto> resultados;

    if (nome == null && descricao == null) {

        resultados = service.listarTodos();

    } else {

        resultados = service.buscarPorFiltros(nome, descricao);

    }

    if (resultados.isEmpty()) {

        return ResponseEntity.noContent().build();

    }

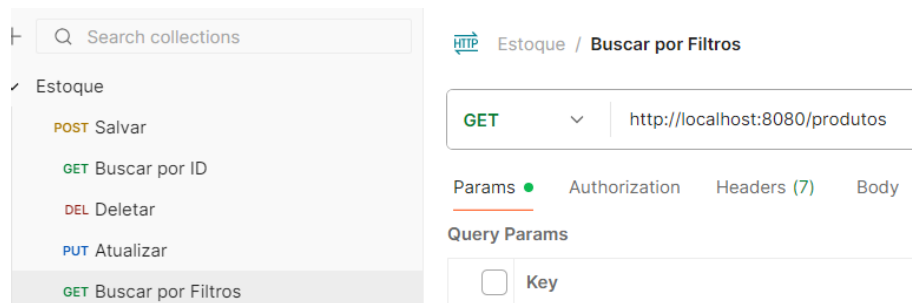
    return ResponseEntity.ok(resultados);
}
```

Os parâmetros nome e descricao são recebidos através da URL, utilizando a anotação @RequestParam. Como estão marcados como required = false, o cliente pode enviar ambos, apenas um ou nenhum deles.

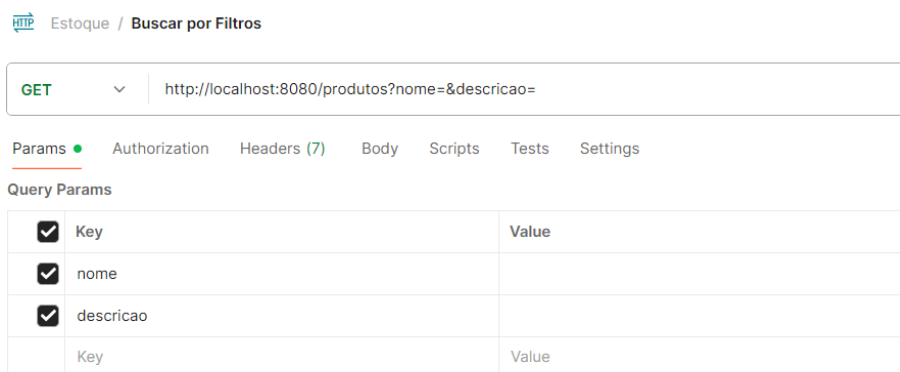
O controller repassa esses parâmetros para a camada de serviço, que executa a busca no banco de dados. O resultado da busca é uma lista de produtos que correspondem aos critérios informados. Por fim, o método retorna a resposta com status 200 OK e a lista de produtos no corpo.

4.9.6.4. Testando Método “Buscar por Filtros” no Postman e verificando no pgAdmin

Para criar esta requisição, daremos o nome de “Buscar por Filtros” no Postman. As configurações serão method = “GET” e url = “http://localhost:8080/produtos”.



Em Params, teremos duas Query Params, uma para nome e uma para descricao. Veja abaixo como ficou:



As Query Params podem ser incluídas pela url, ou na própria aba de Params. Você pode selecionar qual irá usar clicando na caixinha à esquerda, nenhum dos dois parâmetros são obrigatórios, pois em ProdutoController foi passado como não obrigatório dentro do parâmetro de buscarPorFiltros ao usarmos `@RequestParam(required = false)`:

```
@GetMapping
public ResponseEntity<List<Produto>> buscarPorFiltros(@RequestParam(required = false) String nome,
                                                       @RequestParam(required = false) String descricao){
```

Na sequência iremos testar as possibilidades deste método. Para começar, vamos adicionar mais quatro produtos no banco para ajudar no entendimento deste método.

Adicionando o produto de nome = “Mouse” e descricao = “Mouse azul”:

Estoque / Salvar

POST http://localhost:8080/produtos

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL ☒ JSON

```
1 {
2   "nome": "Mouse",
3   "descricao": "Mouse azul",
4   "quantidade": 5,
5   "preco": 100.00
6 }
```

Body Cookies Headers (6) Test Results 201 Created

{ } JSON Preview Visualize

```
1 {
2   "id": 3,
3   "nome": "Mouse",
4   "descricao": "Mouse azul",
5   "quantidade": 5,
6   "preco": 100.00
7 }
```

Agora adicionaremos outro produto de mesmo nome, porém, com descrição diferente:

Estoque / Salvar

POST http://localhost:8080/produtos

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL ☒ JSON

```
1 {
2   "nome": "Mouse",
3   "descricao": "Mouse preto",
4   "quantidade": 5,
5   "preco": 100.00
6 }
```

Body Cookies Headers (6) Test Results 201 Created

{ } JSON Preview Visualize

```
1 {
2   "id": 4,
3   "nome": "Mouse",
4   "descricao": "Mouse preto",
5   "quantidade": 5,
6   "preco": 100.00
7 }
```

O terceiro produto será um Macbook com descricao = “Macbook 32GB 1TB Branco”:

HTTP Estoque / Salvar

POST http://localhost:8080/produtos

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "nome": "Macbook",
3   "descricao": "Macbook 32GB 1TB Branco",
4   "quantidade": 10,
5   "preco": 9000.00
6 }
```

Body Cookies Headers (6) Test Results 201 Created

{ JSON Preview Visualize

```
1 {
2   "id": 5,
3   "nome": "Macbook",
4   "descricao": "Macbook 32GB 1TB Branco",
5   "quantidade": 10,
6   "preco": 9000.00
7 }
```

O quarto produto será um Notebook com descricao = “Acer 32GB 500GB preto”:

HTTP Estoque / Salvar

POST http://localhost:8080/produtos

Params Authorization Headers (9) Body Scripts Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON

```
1 {
2   "nome": "Notebook",
3   "descricao": "Acer 32GB 500GB preto",
4   "quantidade": 12,
5   "preco": 6000.00
6 }
```

Body Cookies Headers (6) Test Results 201 Created

{ JSON Preview Visualize

```
1 {
2   "id": 6,
3   "nome": "Notebook",
4   "descricao": "Acer 32GB 500GB preto",
5   "quantidade": 12,
6   "preco": 6000.00
7 }
```

Agora iremos testar o método de busca por filtro, primeiro com parâmetros vazios.

GET
▼
http://localhost:8080/produtos

Params
Authorization
Headers (7)
Body
Scripts

Query Params

<input type="checkbox"/>	Key
<input type="checkbox"/>	nome
<input type="checkbox"/>	descricao
	Key

Será retornado status “200 OK”, além de todos os produtos existentes no banco.

HTTP Estoque / Buscar por Filtros

GET
▼
http://localhost:8080/produtos

Params
Authorization
Headers (7)
Body
Scripts
Tests
Settings

Body
Cookies
Headers (5)
Test Results
↺
200 OK

{} JSON
Preview
Visualize
▼

```

2      {
3        "id": 1,
4        "nome": "Teclado Mecânico",
5        "descricao": "Teclado sem iluminação RGB",
6        "quantidade": 10,
7        "preco": 110.00
8      },
9      {
10       "id": 3,
11       "nome": "Mouse",
12       "descricao": "Mouse azul",
13       "quantidade": 5,
14       "preco": 100.00
15     },
16     {
17       "id": 4,
18       "nome": "Mouse",
19       "descricao": "Mouse preto",
20       "quantidade": 5,
21       "preco": 100.00
22     },
23     {
24       "id": 5,
25       "nome": "Macbook",
26       "descricao": "Macbook 32GB 1TB Branco",
27       "quantidade": 10,
28       "preco": 9000.00

```

O próximo teste será em relação ao parâmetro “nome”. Este método retornará o produto informado, mesmo que passe apenas parte do nome ou parte da descrição, por exemplo, se for passado o nome “Mou”, ou “se”, ou “ous”, ou qualquer sequência de caracteres que façam parte de um nome ou descrição. Além disso, não haverá distinção de letras minúsculas e maiúsculas. Abaixo temos o teste passando apenas parte do nome, sendo nome = “ous”, a ideia é encontrar os produtos com nome = “Mouse”:

GET ⌵ | http://localhost:8080/produtos?nome=ous

Params ● Authorization Headers (7) Body Scripts Tests Settings

Query Params

<input type="checkbox"/>	Key	Value	Desc
<input type="checkbox"/>	descricao		
<input checked="" type="checkbox"/>	nome	ous	
	Key	Value	Desc

Body Cookies Headers (5) Test Results ⌵

200 OK

{} JSON ⌵ ▶ Preview 🔗 Visualize ⌵

```

1  [
2    {
3      "id": 3,
4      "nome": "Mouse",
5      "descricao": "Mouse azul",
6      "quantidade": 5,
7      "preco": 100.00
8    },
9    {
10     "id": 4,
11     "nome": "Mouse",
12     "descricao": "Mouse preto",
13     "quantidade": 5,
14     "preco": 100.00
15   }
16 ]

```

Pesquisando com nome = “mou”:

GET ⌵ | http://localhost:8080/produtos?nome=mou

Params ● Authorization Headers (7) Body Scripts Tests Settings

Query Params

<input type="checkbox"/>	Key	Value	Descr
<input type="checkbox"/>	descricao		
<input checked="" type="checkbox"/>	nome	mou	
	Key	Value	Descr

Body Cookies Headers (5) Test Results ⌵

200 OK

{} JSON ⌵ ▶ Preview 🔗 Visualize ⌵

```

1  [
2    {
3      "id": 3,
4      "nome": "Mouse",
5      "descricao": "Mouse azul",
6      "quantidade": 5,
7      "preco": 100.00
8    },
9    {
10     "id": 4,
11     "nome": "Mouse",
12     "descricao": "Mouse preto",
13     "quantidade": 5,
14     "preco": 100.00
15   }
16 ]

```

Agora testando com parte do nome e parte da descricao, com descricao = “pre” e nome = “mou”:

HTTP Estoque / Buscar por Filtros

GET

Params **Authorization** Headers (7) Body Scripts Tests Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value
<input checked="" type="checkbox"/>	descricao	pre
<input checked="" type="checkbox"/>	nome	mou
	Key	Value

Body Cookies Headers (5) Test Results

JSON

```
1 [
2   {
3     "id": 4,
4     "nome": "Mouse",
5     "descricao": "Mouse preto",
6     "quantidade": 5,
7     "preco": 100.00
8   }
9 ]
```

Pesquisando por nome = “Macbook”:

HTTP Estoque / Buscar por Filtros

GET

Params **Authorization** Headers (7) Body Scripts Tests Settings

Query Params

<input type="checkbox"/>	Key	Value	Descri
<input type="checkbox"/>	descricao	<input type="text"/>	
<input checked="" type="checkbox"/>	nome	Macbook	
	Key	Value	Descri

Body Cookies Headers (5) Test Results 200 OK

JSON

```
1 [
2   {
3     "id": 5,
4     "nome": "Macbook",
5     "descricao": "Macbook 32GB 1TB Branco",
6     "quantidade": 10,
7     "preco": 9000.00
8   }
9 ]
```

Agora se pesquisarmos por nome = “Macbook” e descricao = “preto”, não encontraremos nada e nosso status retornará “204 No Content”:

HTTP Estoque / Buscar por Filtros

GET http://localhost:8080/produtos?descricao=preto&nome=Macbook

Params Authorization Headers (7) Body Scripts Tests Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value	Descri
<input checked="" type="checkbox"/>	descricao	preto	
<input checked="" type="checkbox"/>	nome	Macbook	
	Key	Value	Descri

Body Cookies Headers (3) Test Results 204 No Content

Raw Preview Visualize

1

Isso porque não existe nenhum Macbook salvo no banco de produtos com a descrição “preto”.

Se buscarmos por “32Gb”, encontraremos o Macbook e o Notebook. Isso porque ambos têm “32Gb” como parte de sua descrição.

HTTP Estoque / Buscar por Filtros

GET http://localhost:8080/produtos?descricao=32Gb&nome=

Params Authorization Headers (7) Body Scripts Tests Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value	Descri
<input checked="" type="checkbox"/>	descricao	32Gb	
<input checked="" type="checkbox"/>	nome		
	Key	Value	Descri

Body Cookies Headers (5) Test Results 200 OK

{ } JSON Preview Visualize

```
1 [
2   {
3     "id": 5,
4     "nome": "Macbook",
5     "descricao": "Macbook 32GB 1TB Branco",
6     "quantidade": 10,
7     "preco": 9000.00
8   },
9   {
10    "id": 6,
11    "nome": "Notebook",
12    "descricao": "Acer 32GB 500GB preto",
13    "quantidade": 12,
14    "preco": 6000.00
15  }
16 ]
```

4.9.7. Remover Quantidade específica de produtos

Endpoint para remover uma quantidade específica de produtos no estoque.

4.9.7.1. Service

O método atualizarQuantidade, dentro da camada Service, é utilizado para atualizar a quantidade de um produto no banco de dados.

```
public Produto atualizarQuantidade(Produto produto) {  
    return repository.save(produto);  
}
```

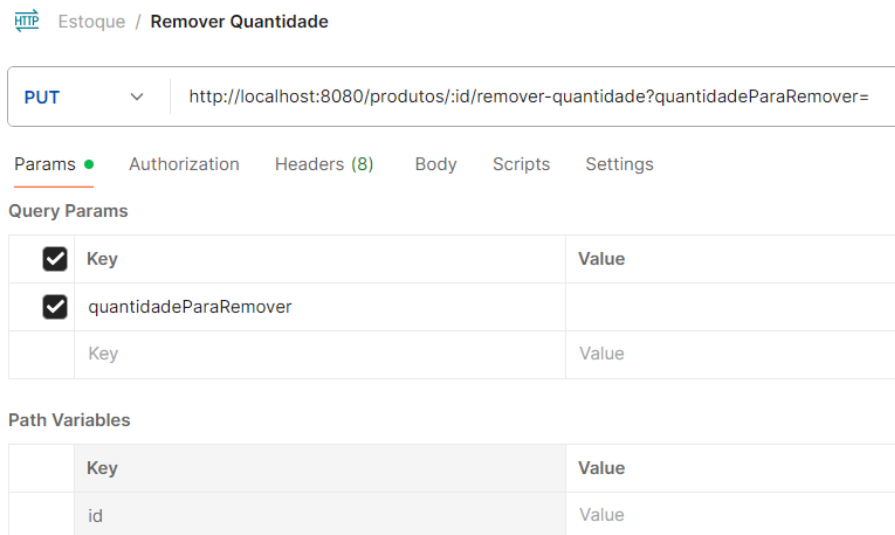
4.9.7.2. Controller

```
@PutMapping("/produtos/{id}/remover-quantidade")  
public ResponseEntity<?> removerQuantidade(@PathVariable Long id, @RequestParam int  
quantidadeParaRemover){  
  
    Optional<Produto> produtoOpt = service.buscarPorId(id);  
    if (produtoOpt.isEmpty()){  
        return ResponseEntity.notFound().build();  
    }  
  
    Produto produto = produtoOpt.get();  
  
    if (quantidadeParaRemover <= 0){  
        return ResponseEntity.badRequest().body("Quantidade para remover deve ser maior que zero.");  
    }  
  
    if (produto.getQuantidade() < quantidadeParaRemover){  
        return ResponseEntity.badRequest().body("Quantidade insuficiente em estoque.");  
    }  
  
    produto.setQuantidade(produto.getQuantidade() - quantidadeParaRemover);  
  
    //se a quantidade ficar zerada, o produto é removido da tabela  
    if (produto.getQuantidade() == 0){  
        service.deletar(id);  
        return ResponseEntity.ok("Produto removido pois a quantidade chegou a zero.");  
    }  
  
    Produto atualizado = service.atualizarQuantidade(produto);  
    return ResponseEntity.ok(atualizado);  
}
```

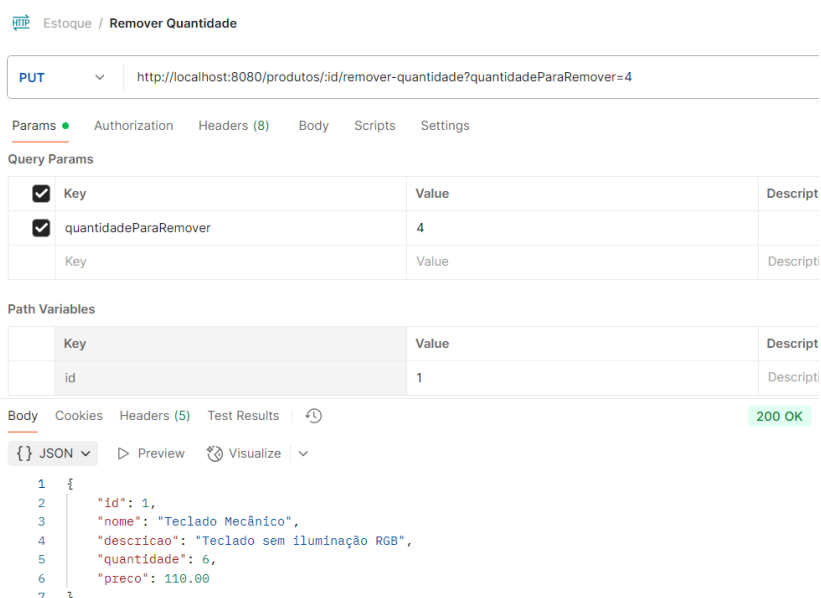
Remove determinada quantidade de um produto e, se essa quantidade chegar a zero, deletar o produto da base de dados.

4.9.7.3. Testando Método “Remover Quantidade” no Postman e verificando no pgAdmin

Agora testaremos o método `removerQuantidade`. Neste método será recebido o Id e a quantidade do produto que desejamos remover. A configuração será como mostrado na imagem:



Para testar este método, foi escolhido o produto de `id = 1`, já a `quantidadeRemovida = 4`. Antes de realizar a remoção temos 10 Teclados Mecânicos cadastrados no banco. Após remover os 4 teremos 6 registrados no banco.



Se tentarmos remover uma quantidade maior do que a quantidade que temos no banco, o sistema não irá permitir e irá retornar uma mensagem dizendo que o estoque é insuficiente

para realizar a quantidade de vendas do produto. Ou seja, se tentarmos remover 7 ou mais produtos, não será possível realizar a ação:

HTTP Estoque / Remover Quantidade

PUT http://localhost:8080/produtos/:id/remover-quantidade?quantidadeParaRemover=7

Params Authorization Headers (8) Body Scripts Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description
<input checked="" type="checkbox"/>	quantidadeParaRemover	7	
	Key	Value	Description

Path Variables

	Key	Value	Description
	id	1	Description

Body Cookies Headers (4) Test Results 400 Bad Request

Raw Preview Visualize

1 Estoque insuficiente para remover a quantidade desejada.

Caso realize a venda com uma quantidade igual ao total presente no banco, o produto é removido do banco. Para isso, removeremos 6 em quantidade, que é o total que temos no momento:

HTTP Estoque / Remover Quantidade

PUT http://localhost:8080/produtos/:id/remover-quantidade?quantidadeParaRemover=6

Params Authorization Headers (8) Body Scripts Settings

Query Params

<input checked="" type="checkbox"/>	Key	Value	Description
<input checked="" type="checkbox"/>	quantidadeParaRemover	6	
	Key	Value	Description

Path Variables

	Key	Value	Description
	id	1	Description

Body Cookies Headers (5) Test Results 200 OK

Raw Preview Visualize

1 Produto removido pois a quantidade chegou a zero.

Após realizar todas essas ações, nosso banco ficou da seguinte forma(caso tenha realizado as ações exatamente igual ao deste livro):

<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div></div>					
	id [PK] bigint	nome character varying (255)	descricao character varying (255)	quantidade integer	preco numeric (38,2)
1	3	Mouse	Mouse azul	5	100.00
2	4	Mouse	Mouse preto	5	100.00
3	5	Macbook	Macbook 32GB 1TB Branco	10	9000.00
4	6	Notebook	Acer 32GB 500GB preto	12	6000.00

Capítulo 5

Código Completo do Projeto

Produto.java

```
package com.estoque.estoque_api.model;

import jakarta.persistence.*;
import lombok.Data;

import java.math.BigDecimal;

@Entity
@Table(name = "produtos")
@Data
public class Produto {

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

    @Column(nullable = false)
    private String nome;

    private String descricao;

    @Column(nullable = false)
    private Integer quantidade;

    @Column(nullable = false)
    private BigDecimal preco;

    public void incrementarQuantidade(int quantidadeASomar) {
        if(quantidadeASomar > 0){
            this.quantidade = this.quantidade + quantidadeASomar;
        }
    }
}
```

Repository.java

```
package com.estoque.estoque_api.repository;

import com.estoque.estoque_api.model.Produto;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
```

```

import java.util.List;
import java.util.Optional;

@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    Optional<Produto> findByNomeAndDescricao(String nome, String descricao);

    @Query("SELECT p FROM Produto p " +
        "WHERE (:nome IS NULL OR LOWER(COALESCE(p.nome, '')) LIKE LOWER(CONCAT('%', :nome, '%')))" +
        "AND (:descricao IS NULL OR LOWER(COALESCE(p.descricao, '')) LIKE LOWER(CONCAT('%', :descricao, '%')))"
    List<Produto> buscarPorFiltros(@Param("nome") String nome, @Param("descricao") String descricao);
}

```

Service.java

```

package com.estoque.estoque_api.service;

import com.estoque.estoque_api.model.Produto;
import com.estoque.estoque_api.repository.ProdutoRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class ProdutoService {
    private final ProdutoRepository repository;

    public Produto salvar(Produto produto) {
        Optional<Produto> existente = repository.findByNomeAndDescricao(produto.getNome(), produto.getDescricao());

        if (existente.isPresent()) {
            Produto produtoExistente = existente.get();
            produtoExistente.incrementarQuantidade(produto.getQuantidade());
            return repository.save(produtoExistente);
        } else {
            return repository.save(produto);
        }
    }

    public List<Produto> listarTodos() {
        return repository.findAll();
    }

    public Optional<Produto> buscarPorId(Long id) {
        return repository.findById(id);
    }
}

```

```

    }

    public List<Produto> buscarPorFiltros(String nome, String descricao) {
        if (nome != null){
            nome = nome.toLowerCase();
        }
        if (descricao != null){
            descricao = descricao.toLowerCase();
        }
        return repository.buscarPorFiltros(nome, descricao);
    }

    public Optional<Produto> atualizar(Long id, Produto produtoAtualizado){
        return repository.findById(id).map(produtoExistente -> {
            produtoExistente.setNome(produtoAtualizado.getNome());
            produtoExistente.setDescricao(produtoAtualizado.getDescricao());
            produtoExistente.setQuantidade(produtoAtualizado.getQuantidade());
            produtoExistente.setPreco(produtoAtualizado.getPreco());
            return repository.save(produtoExistente);
        });
    }

    public void deletar(Long id){
        repository.deleteById(id);
    }

    public Produto atualizarQuantidade(Produto produto) {
        return repository.save(produto);
    }
}

```

Controller.java

```

package com.estoque.estoque_api.controller;

import com.estoque.estoque_api.model.Produto;
import com.estoque.estoque_api.service.ProdutoService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.net.URI;
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/produtos")
@RequiredArgsConstructor
public class ProdutoController {
    private final ProdutoService service;
}

```

```

@PostMapping
public ResponseEntity<Produto> salvar(@RequestBody Produto produto) {
    Produto salvo = service.salvar(produto);
    URI location = URI.create("/produtos/" + salvo.getId());
    return ResponseEntity
        .created(location)
        .body(salvo);
}

@GetMapping("/{id}")
public ResponseEntity<Produto> buscarPorId(@PathVariable Long id) {
    return service.buscarPorId(id)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}

@GetMapping
public ResponseEntity<List<Produto>> buscarPorFiltros(@RequestParam(required = false) String nome,
    @RequestParam(required = false) String descricao){
    List<Produto> resultados;

    if (nome == null && descricao == null) {
        resultados = service.listarTodos();
    } else {
        resultados = service.buscarPorFiltros(nome, descricao);
    }

    if (resultados.isEmpty()) {
        return ResponseEntity.noContent().build();
    }

    return ResponseEntity.ok(resultados);
}

@PutMapping("/{id}")
public ResponseEntity<Produto> atualizar(@PathVariable Long id, @RequestBody Produto produto){
    return service.atualizar(id, produto)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deletar(@PathVariable Long id){
    service.deletar(id);
    return ResponseEntity.noContent().build();
}

@PutMapping("/{id}/remover-quantidade")
public ResponseEntity<?> removerQuantidade(
    @PathVariable Long id,

```

```
    @RequestParam int quantidadeParaRemover) {

    Optional<Produto> produtoOpt = service.buscarPorId(id);
    if (produtoOpt.isEmpty()) {
        return ResponseEntity.notFound().build();
    }

    if (quantidadeParaRemover <= 0) {
        return ResponseEntity.badRequest().body("A quantidade deve ser maior que zero.");
    }

    Produto produto = produtoOpt.get();
    int quantidadeAtual = produto.getQuantidade();

    if (quantidadeParaRemover > quantidadeAtual) {
        return ResponseEntity.badRequest().body("Estoque insuficiente para remover a quantidade desejada.");
    }

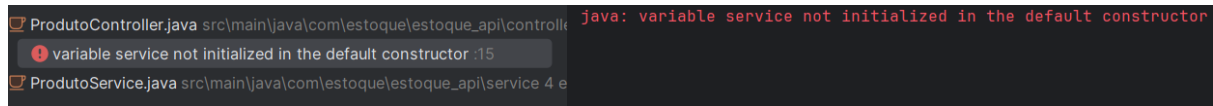
    int novaQuantidade = quantidadeAtual - quantidadeParaRemover;

    if (novaQuantidade == 0) {
        service.deletar(id);
        return ResponseEntity.ok("Produto removido pois a quantidade chegou a zero.");
    } else {
        produto.setQuantidade(novaQuantidade);
        Produto atualizado = service.atualizarQuantidade(produto);
        return ResponseEntity.ok(atualizado);
    }
}
}
```

Capítulo 6

Resolução de alguns possíveis problemas que podem aparecer

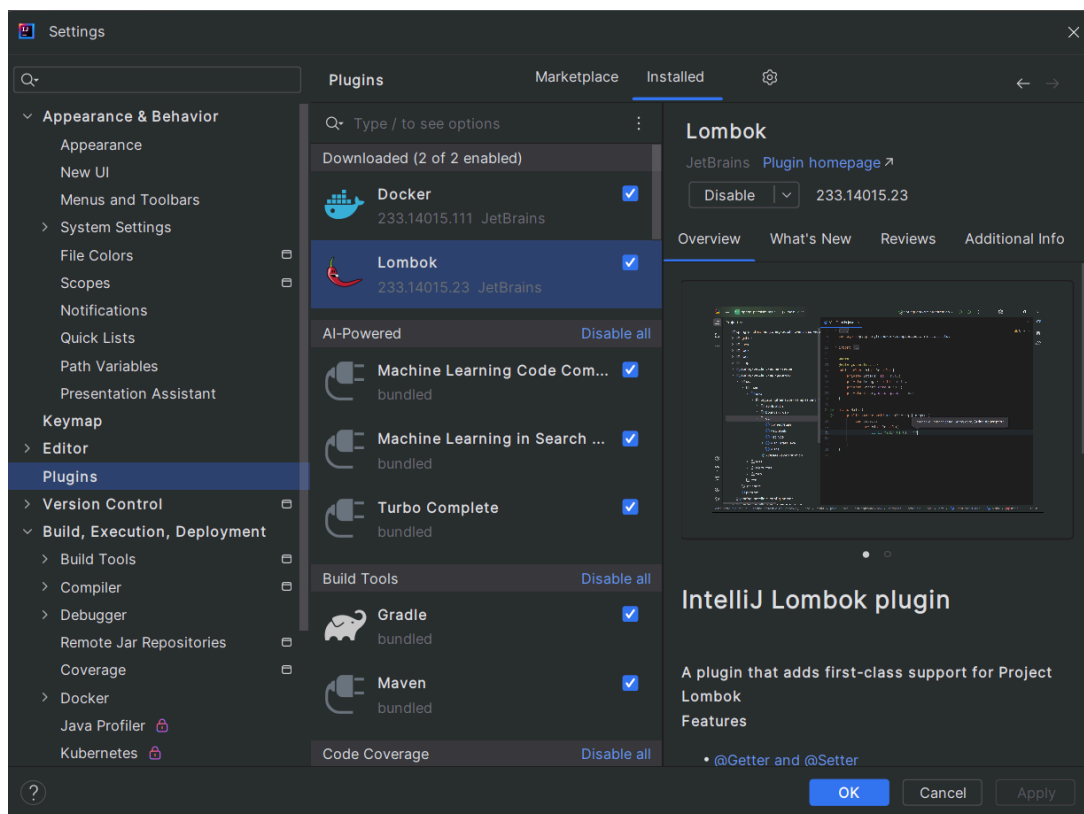
Erro 1: “java: variable service not initialized in the default constructor”



Possível causa: Lombok não está criando os métodos necessários ao usar a anotação `@Data`.

Resolução:

Passo 1: instale o plugin do Lombok



Passo 2: Substitua seu plugin maven-compiler-plugin por este abaixo:

```
<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version> <!-- ou a mais recente -->

<configuration>
```

```
<source>${java.version}</source>

<target>${java.version}</target>

<annotationProcessorPaths>

  <path>

    <groupId>org.projectlombok</groupId>

    <artifactId>lombok</artifactId>

    <version>1.18.30</version> <!-- adicione a versão -->

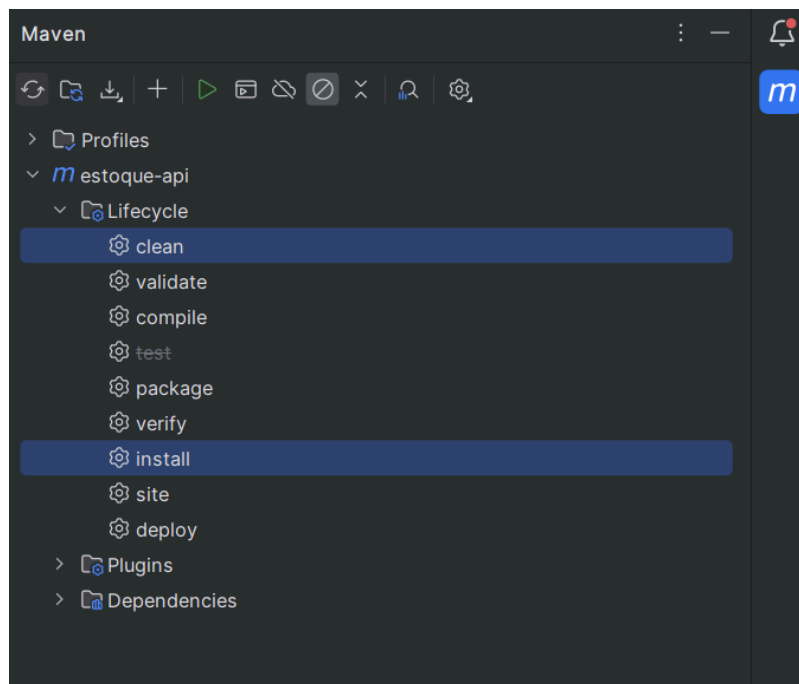
  </path>

</annotationProcessorPaths>

</configuration>

</plugin>
```

Passo 3: selecione Clean, Install e marque a opção de Skip Tests



Agora clique em Run Maven Build.

Sempre que alterar algo no pom.xml ou no .yaml recomendo que realize o Passo 3.

Passo 4: Reinicie a IDE

Capítulo 7

Conclusão

Neste livro, construímos juntos uma API RESTful utilizando o Spring Boot com banco de dados PostgreSQL e testamos cada funcionalidade com o Postman. Passamos por todas as etapas essenciais de um projeto backend: estruturação das camadas, criação de endpoints, persistência de dados e organização do código com boas práticas.

Mesmo sendo um projeto simples, ele representa uma base sólida que pode ser expandida para aplicações reais. Você agora já é capaz de: Criar um CRUD completo com Spring Boot e JPA, integrar com banco de dados PostgreSQL, testar endpoints utilizando o Postman e trabalhar com atualizações parciais, filtros e lógica de negócio separada por camadas.

A proposta foi ensinar de forma direta, prática e acessível, com foco no aprendizado progressivo e na aplicação imediata do conhecimento.

7.1. Próximos passos

Não pare por aqui! O que você aprendeu neste livro serve como base para sistemas maiores e mais robustos. Continue praticando, criando novos projetos, e aos poucos seu domínio sobre back-end crescerá naturalmente. Abaixo deixo indicações do que fazer para dar continuidade ao projeto:

Adicionar Validações com Bean Validation: Implemente validações nos campos da entidade Produto usando anotações como `@NotNull`, `@Size`, `@Positive`, etc.

Trate erros de validação com mensagens personalizadas.

Use a dependência `springdoc-openapi-ui` para gerar uma interface de documentação automática para seus endpoints

Paginação e Ordenação: Adicione suporte para paginar os resultados nas buscas, especialmente quando houver muitos produtos cadastrados.