# API RESTful com Spring Boot 3 - API de Produtos

## Descrição

Este projeto é uma API RESTful para gerenciamento de produtos, construída usando Spring Boot 3. Ele inclui funcionalidades básicas de CRUD (Create, Read, Update, Delete) e utiliza Spring Data JPA para persistência de dados, Spring Validation para validação de entradas, e Spring HATEOAS para navegação de recursos.

## Tecnologias

- **Spring Boot 3**: Framework para criar e configurar a aplicação.
- **Spring WEB MVC**: Por ser WEB e para estrutura de controle e gerenciamento de endpoints RESTful.
- **Spring Data JPA**: Para acesso e manipulação de dados no banco de dados.
- **Spring Validation**: Para validação de dados.
- **Spring HATEOAS**: Para fornecer links e navegação entre recursos na API.

## Tasklist:

### 1. Estrutura Inicial

- [x] Iniciar o projeto com Spring Initializer e baixar o arquivo.
- [x] Criar a estrutura inicial.

### 2. Conectar ao Banco de Dados PostgreSQL

- [x] Criar um servidor no PostgreSQL.
- [x] Definir nome e senha do servidor (ex.: 123).
- [x] Configurar o Host name/address: localhost ou IP.

## 3. Modelagem da Entidade `ProductModel`

- [x] Criar a entidade com as seguintes anotações:
  - `@Entity` : Mapeia a classe para o banco de dados.
  - `@Table(name = "nome_tabela")` : Define o nome da tabela.
  - `implements Serializable` : Permite conversão para bytes.
  - `@Id` : Define o campo como chave primária.
  - `@GeneratedValue(strategy = GenerationType.AUTO)` : Geração automática de IDs.
  - `UUID` : Geração de IDs globais únicos, ideal para sistemas distribuídos.

## 4. Criar `ProductRepository`

- [x] Estender `JpaRepository<Entidade, ChavePrimaria>` :
  - Métodos CRUD predefinidos.
  - Consultas personalizadas.
  - Integração com Spring Data JPA e gerenciamento de transações.
  - `@Repository` (opcional): Define a classe como repositório e trata exceções.

## 5. Criar `RestController`

- [x] Gerenciar requisições HTTP (GET, POST, PUT, DELETE).
- [x] Injeção de dependência com `@Autowired` para `ProductRepository` .

## 6. Implementação de Métodos na API

- [x] **POST**:
  - Criar endpoint para cadastrar produtos com `@PostMapping("/products")` .
  - Validar com `@Valid` e utilizar `BeanUtils.copyProperties` .
  - Retorno: `ResponseEntity<ProductModel>` .
- [x] **GET**:
  - Consultar por ID com `@PathVariable(value = "id")` .
- [x] **DELETE**:
  - Implementar método para remover produtos.

## 7. Implementar HATEOAS

- [x] Adicionar HATEOAS para hipermídia e navegação entre recursos.

## 8. Segurança com Spring Security

- [x] Adicionar dependências de Spring Security e Security Test no `pom.xml`.
- [x] Configurar Spring Security para autenticação via JWT.
- [x] Implementar `SecurityConfiguration`:
  - Desativar CSRF.
  - Configurar autenticação Stateless com tokens.
  - Definir permissões por role (ex.: `ADMIN` para POST em `/products`).
- [x] Criar `AuthenticationController` e DTOs:
  - Login (`/login`) e registro de usuários (`/register`).
  - Hash de senhas com BCrypt.
- [x] Implementar geração e verificação de JWT Tokens:
  - `TokenService` para gerenciamento de tokens.
  - `LoginResponseDTO` para retornar o token JWT.
  - `SecurityFilter` para validação de tokens nas requisições.
- [x] Testar autenticação e proteção de endpoints com Postman.

## 9. Validação de Dados

- [x] Adicionar validações nos DTOs.
- [x] Implementar handler global para tratamento de erros.

## 10. Configurações de Lógica de Negócio e Tratamento de Erros

- [x] Criar `GlobalExceptionsHandler` e configurar exceções globais.
- [x] Impedir a criação de produtos com nomes duplicados.
- [x] Restrição de acesso: apenas `USER` pode fazer GET de produtos.
- [x] Melhorar o retorno de erros de tokens no corpo da requisição.

## 11. Testes com JUnit

- [ ] Escrever testes unitários e de integração.

## 12. Documentação da API

- [ ] Adicionar documentação interativa com Swagger ou OpenAPI.

## 13. Cache

- [ ] Implementar caching para otimizar a performance.

## 14. Performance

- [ ] Usar profiling para identificar gargalos de desempenho.
- [ ] Implementar métodos assíncronos para operações de segundo plano.

## 15. Configuração de Ambiente

- [ ] Usar profiles do Spring para separar configurações de desenvolvimento, teste e produção.

## 16. Melhorias no DTO

- [ ] Otimizar respostas da API com uso de DTOs e projeções.

## 17. Logging

- [ ] Adicionar logging detalhado para monitoramento e depuração.

## 18. Deploy

- [ ] Realizar deploy do backend em plataformas gratuitas como Heroku ou Railway.
- [ ] Configurar variáveis de ambiente e banco de dados na plataforma.
- [ ] Testar a API em ambiente de produção.