

2° EDIÇÃO



SPRING BOOT

DA API REST
AOS MICROSERVICES

MICHELLI BRITO

Sobre o livro

Este livro irá abordar sobre Spring Framework, Spring Boot com API's RESTful e Microservices. Todo o código mostrado ao longo deste livro está disponível e atualizado no Github: <https://github.com/MichelliBrito/springboot-api-ebook>.

Sobre a autora

Arquiteta de Software especialista em Microservices Java com Spring, palestrante e instrutora de treinamentos corporativos. Criadora de um dos maiores canais especializados em Java do Brasil e escritora deste Ebook Spring Boot da API REST aos Microservices já baixado por mais de 10 mil pessoas. Premiada Microsoft MVP 2020 na categoria Developer Technologies. Graduada em Engenharia Química e também em Bacharel em Ciência e Tecnologia pela Universidade Federal de Alfenas - Unifal.

Contatos



https://www.instagram.com/brito_michelli/



<https://www.youtube.com/michellibrito>

Sumário

1.Introdução.....	2
2.Plataforma Spring.....	3
2.1.Spring Framework.....	3
2.2.Inversão de Controle.....	5
2.3.Injeção de Dependência	6
2.4.Core Container.....	7
2.5.Beans	8
2.6.Configurando Beans no Spring	9
2.7.@Bean - Métodos Produtores.....	13
3.Spring Boot.....	14
3.1.Iniciando um projeto Spring Boot	15
3.2. IDE.....	16
3.3. Estrutura do Projeto	18
4.API REST com Spring Boot.....	21
4.1.API REST e RESTful	21
4.1.1 Arquitetura REST	21
4.1.2.Tipos de Representações em REST.....	22
4.1.3.Recursos, Métodos e Retornos	23
4.1.4.Modelo Maturidade Richardson	23
4.2.Criando uma API REST com Spring Boot	26
4.2.1.Criando conexão com banco de dados.....	26
4.2.2.Criando o Model e o Repository	26
4.2.3.Criando o Controller	29
4.2.4.Implementando os métodos GET ALL e GET ONE	30
4.2.5.Implementando os métodos POST, DELETE e PUT	32
4.2.6.Inserindo as HATEOAS	35
5.Microservices com Spring Boot.....	39
5.1.Arquitetura de Microservices.....	39
5.2.Comunicação entre Microservices.....	42
5.3.Exemplo de Microservices com Spring Boot.....	43

1.Introdução

Muitas pessoas que estão começando na área de programação ou buscam migrar de tecnologia ou linguagem perguntam se hoje em dia ainda vale a pena estudar e investir na linguagem Java. A resposta para essa pergunta é que vale a pena sim, pois há uma grande demanda desses profissionais no mercado de trabalho, principalmente quando se trata de Java para web utilizando frameworks como o Spring por exemplo e também utilizando arquitetura de microservices com Spring Boot.

Mesmo o Java sendo uma linguagem utilizada já a bastante tempo e bem robusta ainda continua sendo muito utilizada principalmente com o auxílio de frameworks e plataformas bem completas e atualizadas que trazem de forma mais simples e padronizada projetos capazes de impulsionar o desenvolvimento de forma bem mais produtiva, como é o caso do Spring.

Para que o profissional ou iniciante na carreira tenha uma base dos conceitos e práticas necessárias para atuar com essas tecnologias, neste livro será abordado primeiramente a plataforma Spring com os principais conceitos para utilizar o Spring Framework e o Spring Boot. Em seguida, será apresentado os conceitos e padrões da arquitetura REST e mostrado na prática como criar uma API REST com Spring Boot seguindo os principais princípios e constraints, envolvendo os métodos HTTP, recursos, retornos, HATEOAS entre outros.

Também será apresentado o conceito da arquitetura de microservices e a comunicação entre esses serviços e como hoje em dia esse tipo de arquitetura está sendo construída utilizando o Spring Boot.

2. Plataforma Spring

O Spring consiste em uma plataforma completa de recursos para construção de aplicativos Java, que veio para simplificar o desenvolvimento em Java EE com diversos módulos que auxiliam na construção de sistemas reduzindo muito o tempo de desenvolvimento.

Essa plataforma conta com recursos avançados que abrangem várias áreas de uma aplicação com projetos/módulos prontos para uso, como:

- Spring Framework;
- Spring Boot;
- Spring Web;
- Spring Security;
- Spring Data;
- Spring Batch;
- Spring Cloud;
- outros.

Para visualizar e conhecer todos os projetos disponíveis na plataforma Spring, basta acessar o site Spring by Pivotal e conferir a lista completa através do link <https://spring.io/projects>.

Com esses projetos é possível construir aplicativos com funcionalidades avançadas e com uma produtividade muito maior, podendo focar mais nas regras de negócios e deixar as configurações de baixo nível por conta do Spring.

2.1. Spring Framework

O Spring Framework é o projeto base para todo o ecossistema Spring e é dividido em 7 grupos:

- Core Container;
- Data Access/Integration;
- Web;
- Aspect Oriented Programming (AOP);
- Instrumentation;
- Messaging;
- Test.

Dentro do grupo Core Container, estão os módulos responsáveis por conter as partes fundamentais do framework, como as classes básicas e avançadas, suas implementações e controle das definições em tempo de execução das configurações por anotações ou arquivos XML.

O módulo Data Access/Integration é o responsável por prover funcionalidades para transações com o banco de dados. O módulo Web contém os recursos para uma aplicação web, como a implementação do MVC, Web Services REST, entre outros.

O módulo AOP fornece a implementação para programação orientada a aspectos, o módulo Instrumentation fornece implementações de instrumentação e por fim o módulo Messaging contém implementação e suporte para programação baseada em mensagens.

O módulo Test possui o suporte para os testes unitários utilizando JUnit e testes de integração.

Todos os módulos citados acima, com exceção do módulo Test, são construídos sobre o Core Container do Spring Framework, em destaque na imagem abaixo.

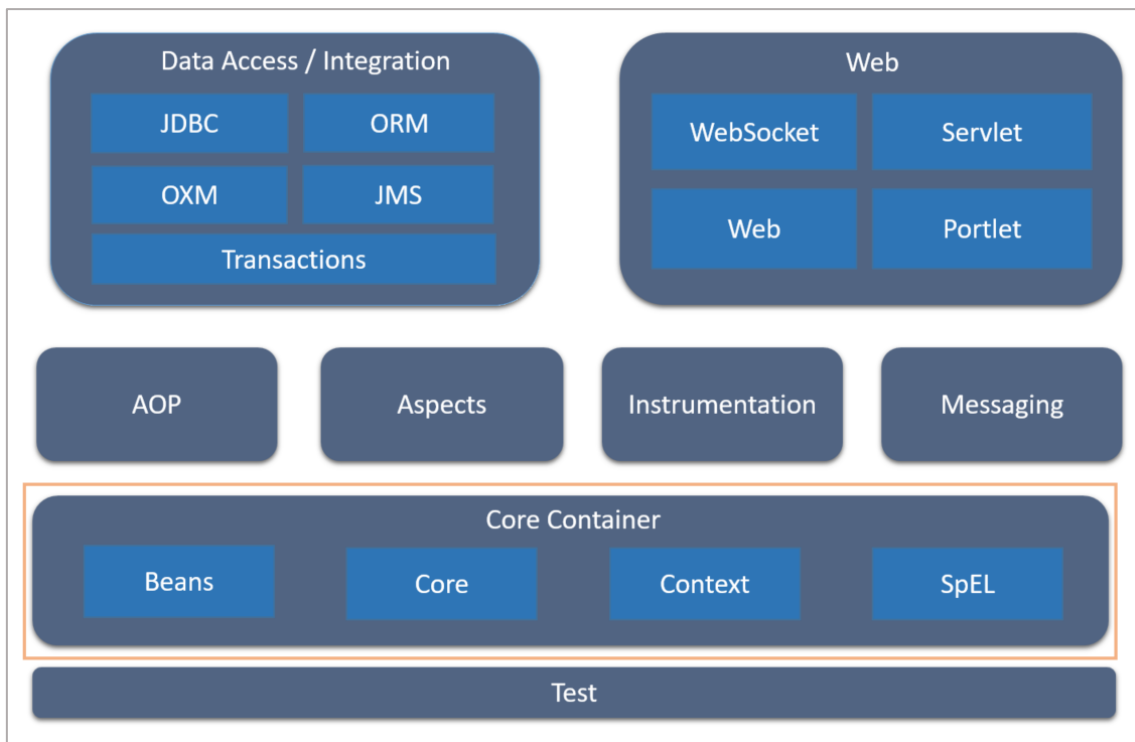


Figura 1: Projeto Spring Framework destacando o módulo Core Container

Como o projeto Spring Framework possui o módulo Core Container, onde está implementado a Inversão de Controle que utiliza da Injeção de Dependência, ele se torna o projeto essencial para iniciar uma aplicação sendo assim a base de toda a plataforma Spring.

Talvez esse último parágrafo tenha ficado um pouco difícil de entender diante dessas terminologias. Primeiramente é preciso entender que todo framework é a aplicação de uma Inversão de Controle. Mas no que consiste essa Inversão de Controle afinal?

2.2. Inversão de Controle

Inversão de Controle (IoC - Inversion of Control) é um processo em que um objeto define suas dependências sem criá-los. Este objeto delega a tarefa de construir tais dependências para um contêiner IoC.

Por exemplo, vamos considerar que temos duas classes, A e B, onde a classe A possui uma dependência da classe B, já que ela utiliza um método de B. Assim, a classe A sempre teria que criar uma nova instância da classe B para que assim pudesse utilizar seu método, como mostra na imagem abaixo.

```
public class A{  
    private B b;  
    public void metodoA() {  
        b = new B();  
        b.metodoB();  
        ...  
    }  
}  
  
public class B{  
    public void metodoB(){  
        ...  
    }  
}
```

Figura 2: Exemplo de uma aplicação que não utiliza IoC (Inversão de Controle)

Porém, quando se utiliza a Inversão de Controle, a classe A não precisa se preocupar em criar uma instância de B, pois essa responsabilidade passa a ser do container do Spring Framework que realiza essa Inversão de Controle através da Injeção de Dependência. E o que seria a Injeção de Dependência afinal?

2.3. Injeção de Dependência

A Injeção de Dependência consiste na maneira, ou seja, na implementação utilizada pelo Spring Framework de aplicar a Inversão de Controle quando for necessário.

A Injeção de Dependência define quais classes serão instanciadas e em quais lugares serão injetadas quando houver necessidade. Assim, basta que a classe A crie um ponto de injeção da classe B, pelo construtor por exemplo, e quando houver a necessidade o container do Spring Framework irá criar uma instância

da classe B para que a classe A possa utilizar o método `b.metodoB()`, como mostra na imagem abaixo.

```
public class A{  
    private B b;  
    public A(B b) {  
        this.b = b;  
    }  
    public void metodoA() {  
        b.metodoB();  
        ...  
    }  
}  
  
public class B{  
    public void metodoB(){  
        ...  
    }  
}
```

Figura 3: Exemplo de Injeção de Dependência

2.4.Core Container

O Spring Framework utiliza da Injeção de Dependência para aplicar a Inversão de Controle no sistema e toda essa implementação está presente no Core Container, onde fica a base de configuração do Spring Framework.

Quando a aplicação é executada, o Core Container é iniciado, as configurações da aplicação pré-definidas em classes ou arquivos XML são lidas e as dependências necessárias são definidas e criadas através da IoC e destruídas quando não mais forem utilizadas. Essas dependências são denominadas beans dentro do contexto do Spring, que consistem em objetos os quais possuem seu ciclo de vida gerenciado pelo container de IoC/ID do Spring. Esses passos definem o ciclo de vida de um Container, como pode ser mostrado também na imagem abaixo.

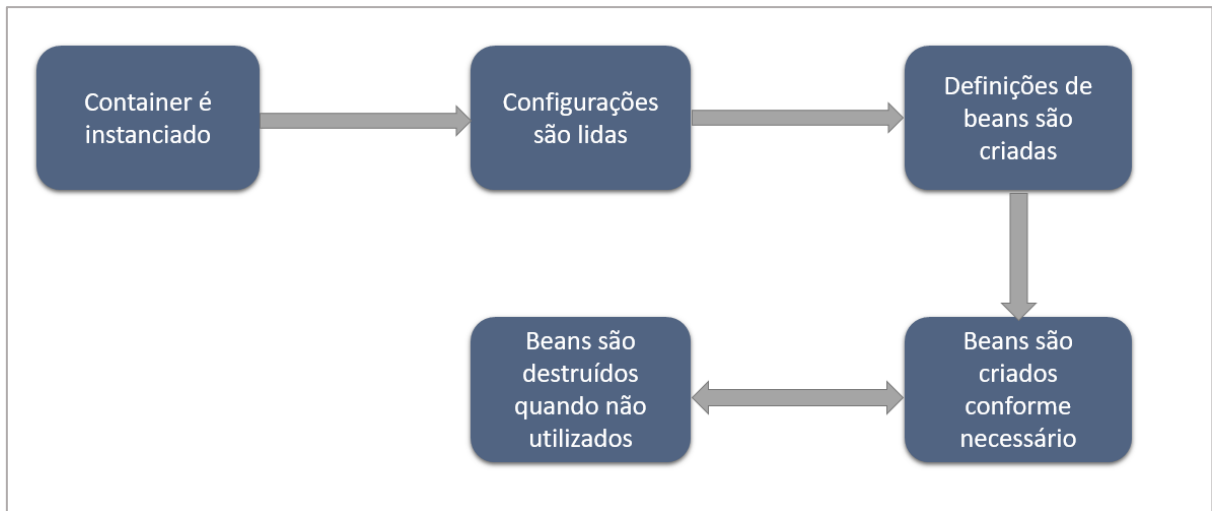


Figura 4: Ciclo de vida com Container

2.5.Beans

Como foi dito anteriormente, um bean consiste em um objeto que é instanciado, montado e gerenciado por um contêiner do Spring através da Inversão de Controle (IoC) e Injeção de Dependências.

Assim como o container, um bean também tem seu ciclo de vida, o qual é iniciado e criado pelo container, as dependências desse bean são injetadas, o método de inicialização é chamado e então, o bean assim é enviado para o cliente, no caso a classe que possui essa dependência, para ser utilizado e em seguida descartado.

Na prática, utilizando o exemplo anterior, quando o container é instanciado ele cria uma instância da classe B, chama o construtor da classe A para injetar esse bean e em seguida, a classe A utiliza esse bean através de `b.metodoB()`. Esse bean então é descartado quando não mais utilizado e tal ciclo pode ser visualizado na imagem abaixo.

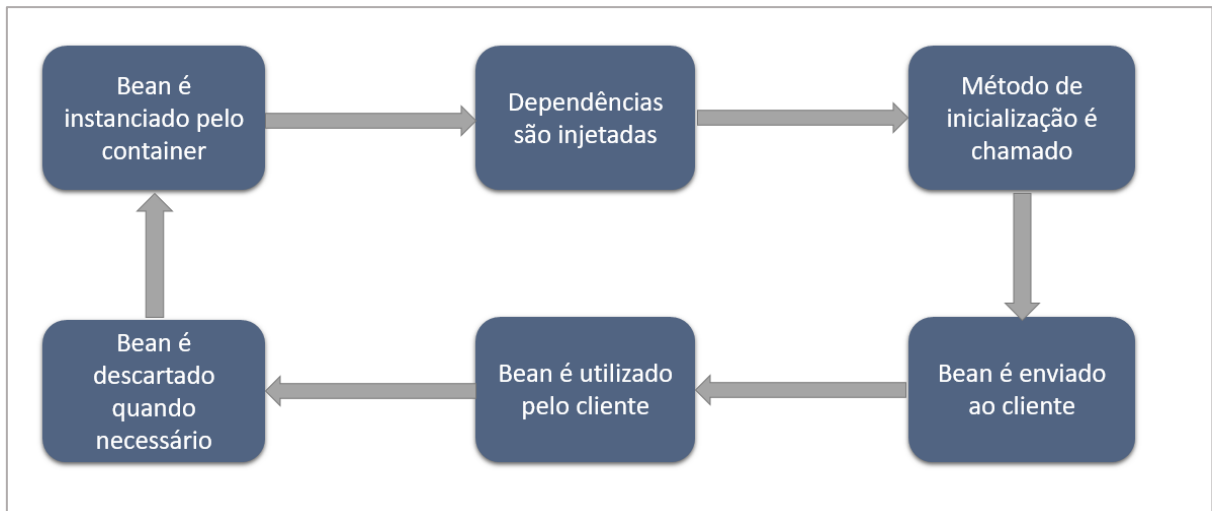


Figura 5: Ciclo de vida de um Bean

2.6. Configurando Beans no Spring

É preciso que o Spring conheça quais as classes da aplicação serão beans gerenciados por ele para que então seja aplicada a IoC/ID. Para isso há duas maneiras de configurar e determinar esses beans, utilizando configurações em arquivos XML ou através de anotações.

Na configuração por XML, não muito utilizada hoje em dia, é preciso definir tags <bean> dentro de uma tag principal <beans> passando o path da classe, assim o Spring saberá quais classes ele irá gerenciar a criação de instâncias e a injeção de suas dependências.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="com.example.springboot.Produto"/>
  <bean class="com.example.springboot.ProdutoController"/>
  <bean class="com.example.springboot.ProdutoService"/>
  <bean class="com.example.springboot.ProdutoRepository"/>
</beans>
  
```

Figura 6: Configuração dos beans por xml

Na configuração por anotações, é possível utilizar os estereótipos do Spring para determinar de forma mais objetiva e específica qual o tipo de bean será cada classe. Há quatro principais tipos:

- @Component;
- @Service;
- @Controller;
- @Repository.

Assim, ao anotar determinada classe com algum desses estereótipos, o Spring entende que tal classe é um bean e será gerenciada por ele. Abaixo seguem alguns exemplos de beans utilizando configuração por anotações.

```
@Component
public class Produto {

    private String nome;
    private BigDecimal valor;

    //... métodos getters e setters

}
```

Figura 7: Exemplo de bean do tipo Component

```
@Service
public class ProdutoService {

    //regras de negócio para Produto

}
```

Figura 8: Exemplo de bean do tipo Service

```
@Controller
public class ProdutoController {

    //... métodos GET, POST
    // DELETE e UPDATE

}
```

Figura 9: Exemplo de bean do tipo Controller

```
@Repository
public class ProdutoRepository {

    //métodos de transação com
    // o banco de dados

}
```

Figura 10: Exemplo de bean do tipo Repository

Considerando que os beans gerenciados pelo Spring já foram definidos a próxima questão é entender como o Spring saberá onde injetar as instâncias que ele irá criar com suas dependências. Para isso é preciso criar os pontos de injeção, que consistem em uma maneira de entregar as dependências ao objeto que necessita. Os dois tipos de pontos de injeção mais comuns são os construtores e setters, os quais podem ser visualizados abaixo.

```
@Service
public class ProdutoService {

    private ProdutoRepository produtoRepository;

    public ProdutoService(ProdutoRepository produtoRepository) {
        super();
        this.produtoRepository = produtoRepository;
    }

    //regras de negócio

}
```

Figura 11: Ponto de Injeção pelo método Construtor

```
@Service
public class ProdutoService {

    private ProdutoRepository produtoRepository;

    public void setProdutoRepository(ProdutoRepository produtoRepository) {
        this.produtoRepository = produtoRepository;
    }

}
```

Figura 12: Ponto de Injeção pelo método Setter

Dentro do Spring, há uma outra maneira de se criar pontos de injeção de forma automática, utilizando a anotação `@Autowired`.

```
@Repository
public class ProdutoRepository {

    //métodos de transação com
    // o banco de dados

}

@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    //regras de negócio

}
```

Figura 13: Ponto de Injeção utilizando `@Autowired`

O ciclo de vida de um bean depende do seu escopo, que pode ser determinado no Spring através da anotação `@Scope`. Os tipos de escopos do Spring utilizados para web são:

- Singleton;
- Prototype;
- Request;
- Session.

Assim, é possível determinar se um bean será do tipo singleton por exemplo, ou seja, o container irá criar uma única instancia desse bean que será utilizada para todas as solicitações da instancia.

Se o bean for configurado como prototype o container irá criar várias instancias, uma para cada solicitação. O bean com escopo do tipo request terá uma instancia criada para cada solicitação HTTP e por fim, o bean com escopo session terá sua instancia preservada e utilizada pelas solicitações enquanto durar a sessão.

2.7.@Bean - Métodos Produtores

Quando é preciso que uma classe externa da aplicação seja um bean gerenciado pelo Spring, é preciso primeiramente criar um método produtor dentro da classe de configuração do Spring, que irá retornar tal classe externa que será gerenciada pelo container do Spring. E para configurar essa classe como sendo um bean, é preciso anotar tal método com `@Bean`, assim é possível criar pontos de injeção e o Spring irá controlá-la como sendo qualquer outro bean dentro da aplicação.

```
@Configuration
public class AppConfig {

    @Bean
    public ArquivoExterno arquivoExterno() {
        ArquivoExterno arquivoExterno = new ArquivoExterno();
        return arquivoExterno;
    }

}

@Service
public class ProdutoService {

    @Autowired
    private ProdutoRepository produtoRepository;

    @Autowired
    private ArquivoExterno arquivoExterno;

    //regras de negócio

}
```

Figura 14: Exemplo de método produtor com `@Bean`

3.Spring Boot

Iniciar uma aplicação do zero utilizando o Spring Framework pode ser um tanto quanto trabalhosa, pois é preciso fazer várias configurações em arquivos XML ou classe de configuração, configurar o Dispatcher Servlet, gerar um arquivo war, subir a aplicação dentro de um Servlet Container, como por exemplo o Tomcat, para então conseguir executar a aplicação e começar a implementar as regras de negócio.

O Spring Boot veio como uma extensão do Spring, que utiliza da base do Spring Framework para iniciar uma aplicação de uma forma bem mais simplificada, diminuindo a complexidade de configurações iniciais e o tempo para executar uma aplicação e deixá-la pronta para implementação das regras de negócio. Também já traz um servidor embutido que facilita ainda mais esse processo de start da aplicação.

Ao iniciar um projeto Spring Boot, basta uma dependência no arquivo pom.xml, `spring-boot-starter`, para que ele já traga internamente todas as dependências base do Spring Framework, como pode-se observar na imagem abaixo.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figura 15: Dependências iniciais de um projeto Spring Boot



Figura 16: Módulos do Spring Core Container – Base do Framework

Resumindo, o Spring Boot é a soma do Spring Framework com um servidor embutido menos as configurações XML e classes de configurações.

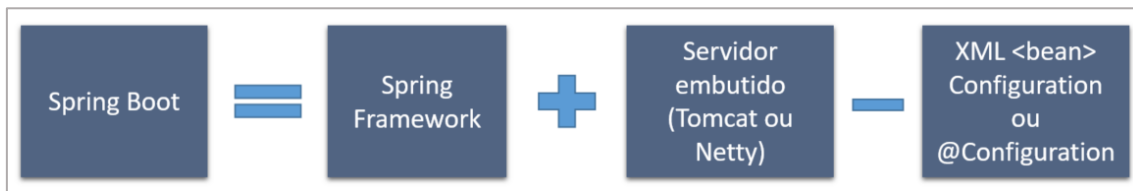


Figura 17: Resumo Spring Boot

3.1. Iniciando um projeto Spring Boot

Para iniciar um projeto Spring Boot leva-se apenas poucos minutos, utilizando o Spring Initializr através do site <https://start.spring.io/>. Basta definir algumas configurações iniciais como o gerenciador de dependências a ser utilizado, a linguagem e versão da linguagem, a versão do Spring Boot, dados básicos do projeto como Group e Artifact e por fim, selecionar as dependências iniciais que irão compor o projeto. Assim, ao clicar no botão Generate um projeto compactado é disponibilizado. Então, é preciso descompactar esse projeto em um determinado diretório na máquina e importá-lo para a IDE escolhida para o desenvolvimento.

Na imagem abaixo é possível visualizar a geração de um projeto Spring Boot através do Spring Initializr, determinando o Maven como gerenciador de dependências, o Java como linguagem de programação, a versão do Spring Boot

como 2.4.5, Group como com.example e Artifact como springboot e selecionando a primeira dependência do projeto: Spring Web.

The screenshot shows the Spring Initializr web application in a browser window. The interface is divided into several sections:

- Project:** Radio buttons for Maven Project (selected) and Gradle Project.
- Language:** Radio buttons for Java (selected) and Kotlin. Below them are radio buttons for Groovy.
- Spring Boot:** Radio buttons for versions: 2.5.0 (SNAPSHOT), 2.5.0 (RC1), 2.4.6 (SNAPSHOT), 2.4.5 (selected), 2.3.11 (SNAPSHOT), and 2.3.10.
- Project Metadata:**
 - Group:** com.example
 - Artifact:** springboot
 - Name:** springboot
 - Description:** Demo project for Spring Boot
 - Package name:** com.example.springboot
 - Packaging:** Radio buttons for Jar (selected) and War.
 - Java:** Radio buttons for 16, 11, and 8 (selected).
- Dependencies:** A section with a button "ADD DEPENDENCIES... CTRL + B". Below it, "Spring Web" is listed with a "WEB" tag and a description: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE..."

Figura 18: Spring Initializr para iniciar um projeto Spring Boot

3.2. IDE

Há várias IDE's que podem ser utilizadas para desenvolver projetos Java com Spring, dentre as mais conhecidas estão o Eclipse, IntelliJ IDEA, Netbeans e o STS (Spring Tool Suite 4), sendo essa última uma IDE baseada no Eclipse própria para projetos Spring, a qual será utilizada nos exemplos ao longo desse livro.

Mas para quem prefere utilizar uma IDE como o Visual Studio Code ou Theia é possível também baixar e instalar o Spring Tool de acordo com a IDE de melhor preferência. Para fazer o download e instalar o Spring Tool para uma dessas IDE's citadas anteriormente, basta acessar o site da Spring by Pivotal através do link <https://spring.io/tools>.

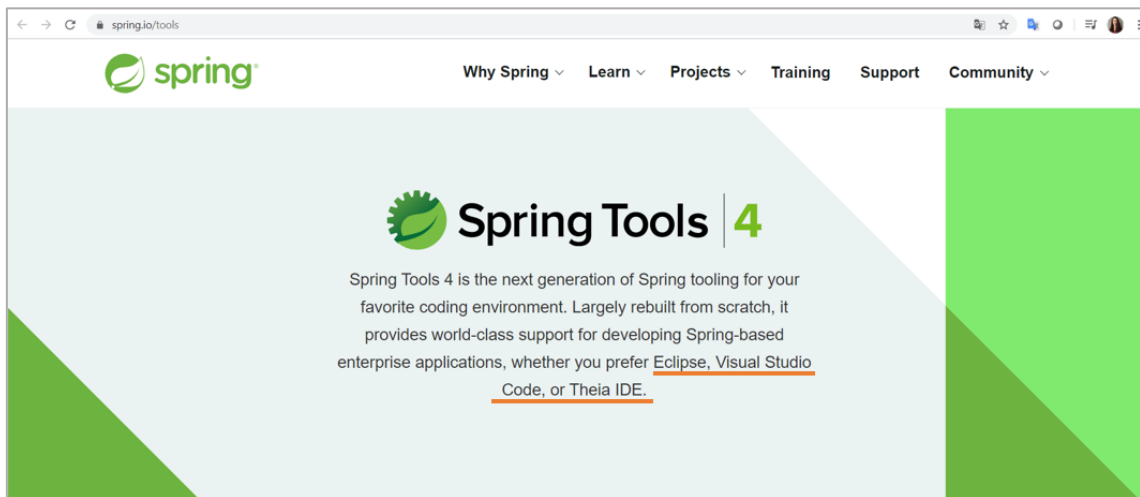


Figura 19: Site Spring Tools

A interface da IDE STS 4 após baixada e instalada pode ser visualizada abaixo, sendo essa bem parecida com o próprio Eclipse.

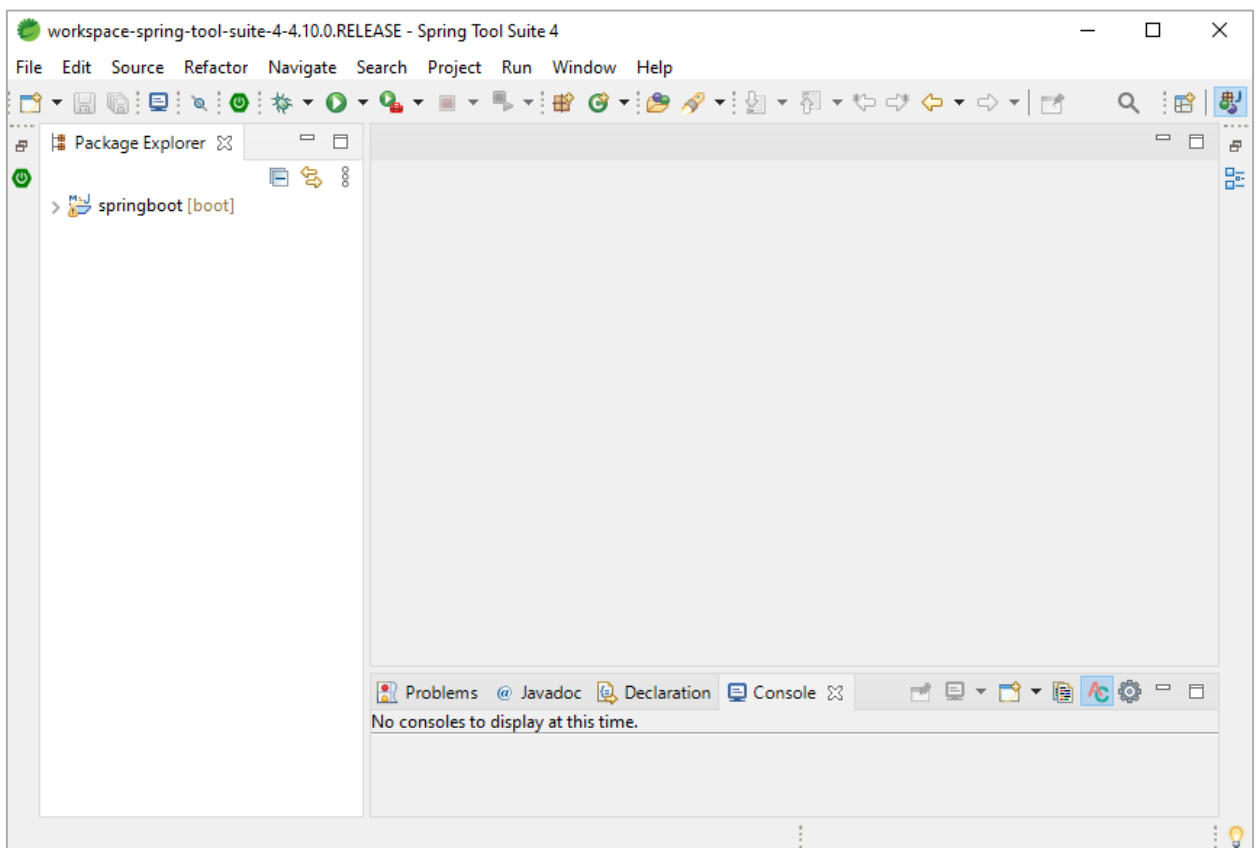


Figura 20: Interface STS

3.3. Estrutura do Projeto

Após descompactar o projeto criado com Spring Initializr, é preciso importá-lo para dentro da IDE escolhida, neste caso, o STS. A estrutura do projeto inicialmente pode ser observada na imagem abaixo.

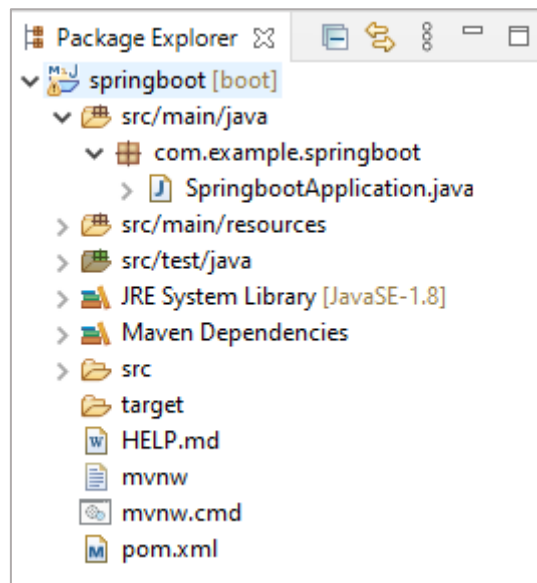


Figura 21: Estrutura inicial de um projeto Spring Boot

Dentro do diretório raiz `com.example.springboot` fica localizada a classe principal da aplicação, `SpringbootApplication`, gerada automaticamente pelo Spring Boot sendo a responsável por inicializar a aplicação. Basta clicar com o botão direito do mouse, clicar em `Run As => 3 Spring Boot App` que a aplicação será iniciada.

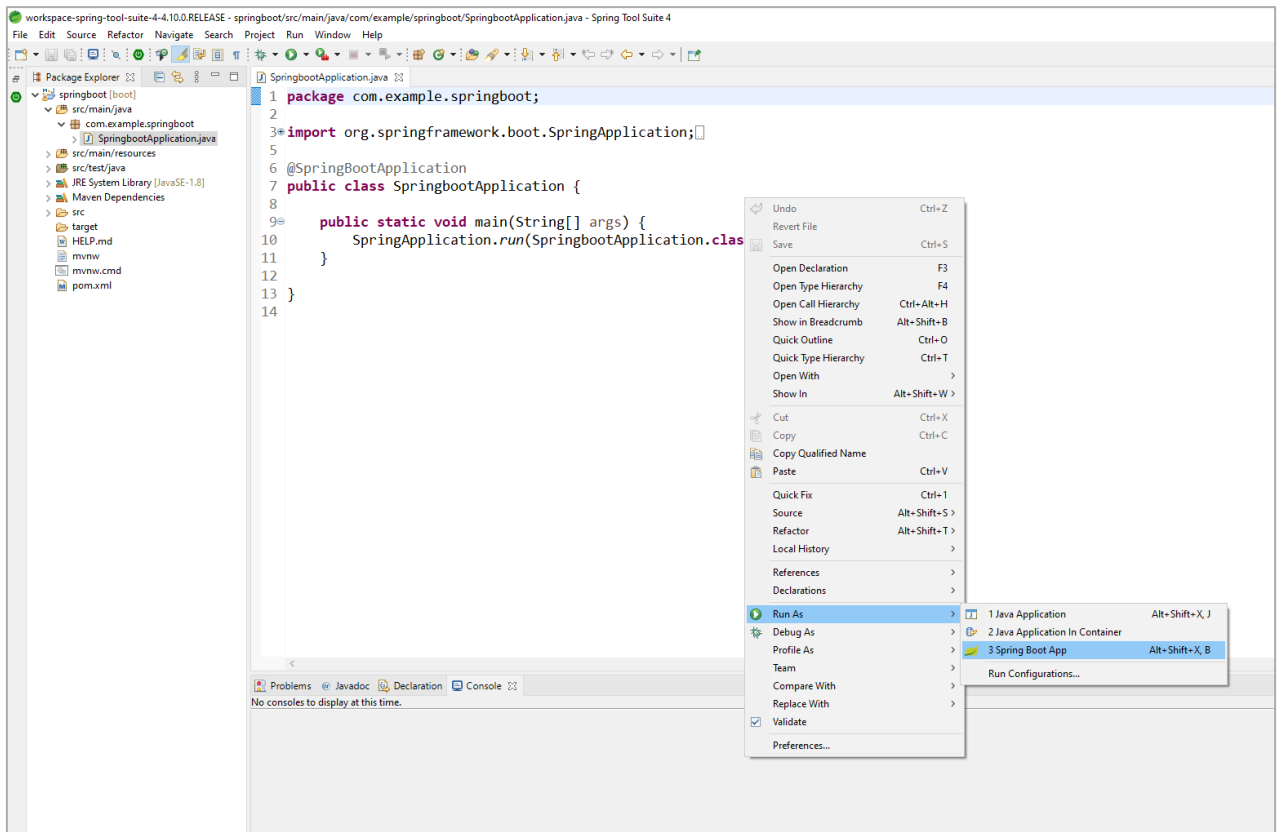


Figura 22: Executando a aplicação

Ao executar a aplicação através da classe SpringbootApplication, um Tomcat embutido é inicializado, deixando a aplicação disponível na porta 8080 neste caso, como mostra na imagem abaixo.



Figura 23: Aplicação disponível na porta 8080

Apenas com esses passos mencionados anteriormente já foi possível criar um projeto Spring Boot e iniciar a aplicação utilizando o STS como IDE. Por isso o

Spring Boot vem sendo tão utilizado pelas empresas, pois reduz drasticamente o tempo de start de um projeto, já que disponibiliza muitas das configurações iniciais pré-definidas e prontas para uso.

4.API REST com Spring Boot

4.1.API REST e RESTful

API REST é uma aplicação cliente/servidor que envia e recebe dados através do protocolo HTTP utilizando de padrões de comunicação como XML e JSON e pode ser implementada na linguagem desejada. Uma API é desenvolvida para permitir a interoperabilidade entre aplicações, por exemplo, considerando dois sistemas, um desktop e um outro mobile, ambos podem consumir a mesma API para montar suas interfaces, ou seja, a mesma API pode ser utilizada por diferentes sistemas.

Uma API pode ser considerada RESTful quando ela utiliza em sua implementação o conceito arquitetural REST. REST é algo abstrato, como um modelo arquitetural enquanto que RESTful é algo mais concreto, como a implementação deste modelo em alguma API. Então, para criar uma API RESTful é preciso conhecer a arquitetura REST e também aplicá-la corretamente.

4.1.1 Arquitetura REST

O modelo arquitetural REST nada mais é que um conjunto de boas práticas para que determinado aplicativo possa ser construído e documentado de uma maneira padronizada, fácil e que gere maior produtividade tanto para os desenvolvedores na construção, quanto para o consumo desse aplicativo por outros sistemas.

Um dos idealizadores desse modelo arquitetural foi Roy Fielding, que definiu em sua tese de doutorado seis constraints, ou seja, regras que devem ser obrigatoriamente seguidas para uma aplicação ser considerada RESTful.

A primeira constraint diz que uma aplicação, neste caso uma API, deve ser cliente/servidor a fim de separar as responsabilidades. Já a segunda constraint diz que essa aplicação deve ser stateless, ou seja, não guardar estado no servidor, para que cada requisição que o cliente envia ao servidor tenha informações relevantes e únicas para a resposta, assim independe qual servidor irá responder esse cliente caso a aplicação esteja escalada em múltiplos servidores, garantindo escalabilidade e disponibilidade.

A terceira constraint define que a aplicação deve ter a capacidade de realizar cache para reduzir o tráfego de dados entre cliente e servidor. A quarta constraint diz que a aplicação deve ter uma interface uniforme, onde sua modelagem deve conter recursos bem definidos, apresentar hipermídias, utilizar corretamente os métodos HTTP e códigos de retorno.

A definição da quinta constraint diz que o sistema deve ser construído em camadas, ou seja, a possibilidade de escalar a aplicação em múltiplos servidores. E por fim, a última constraint diz que a aplicação deve ter a capacidade de evoluir sem a quebra da mesma, ou seja, código sob demanda.

4.1.2. Tipos de Representações em REST

Uma API REST pode utilizar 2 padrões de comunicação, XML e JSON. O padrão XML se baseia em tags, sendo um pouco mais pesado quando comparado ao JSON.

```
<produto>  
  <codigo>1</codigo>  
  <nome>Samsung S20</nome>  
  <valor>4.500,00</valor>  
</produto>
```

Já o padrão JSON trabalha com chave/valor.

```
{  
  "codigo": 1,  
  "nome": "Samsung S20",  
  "valor": 4.500,00  
}
```

4.1.3. Recursos, Métodos e Retornos

Um recurso é utilizado para identificar de forma única um objeto abstrato ou físico e é representado pela URI da API. Tal recurso como é um objeto deve ser representado por um substantivo e nunca por um verbo.

recursos: [/produtos](#)
[/produtos/1](#)

Os métodos utilizados na construção de uma API REST são os métodos do protocolo HTTP. Para obter um determinado recurso por exemplo utiliza-se o método GET, o método POST para criar um novo recurso, o método PUT para atualizar esse recurso criado e o DELETE para deletar tal recurso. Há outros métodos HTTP que podem ser utilizados, mas esses citados anteriormente são os mais comuns.

Os retornos são os códigos e as respostas que o servidor retorna ao cliente diante de uma requisição. Os principais são:

- 1XX - Informações;
- 2XX - Sucesso;
- 3XX - Redirecionamento;
- 4XX - Erro no cliente;
- 5XX - Erro no servidor.

4.1.4. Modelo Maturidade Richardson

Existem casos que é preciso seguir uma abordagem menos robusta para a construção de uma API, e seguir todas as seis constraints propostas por Roy Fielding pode ficar inviável. Pensando nesse ponto de vista, para casos mais simples, Leonard Richardson propôs um modelo de maturidade composto por quatro níveis e a API que alcançar esses quatro níveis pode sim ser considerada uma API RESTful.

Uma API atinge o nível 0, também chamado de nível POX, quando utiliza o protocolo HTTP apenas como mecanismo de comunicação e não como semântica de seus verbos. Quando uma API atinge o nível 1, ela utiliza os recursos de maneira correta, definindo bem e de forma única cada recurso e utilizando os substantivos para representar os objetos.

Uma API no nível 2 utiliza o protocolo HTTP de forma semântica com seus métodos e também define os tipos de retorno para cada resposta possível de uma requisição. E por fim, uma API que atinge o nível 3 possui as HATEOAS que são as hipermídias que mostram o seu estado atual e seu relacionamento com os elementos ou estados futuros, ou seja, a capacidade de um documento se relacionar com os demais.

No exemplo abaixo, é possível visualizar uma API que segue todos esses níveis, utiliza o método GET HTTP de forma semântica com retorno 200 OK, tem seu recurso bem definido, como `/produtos` e `/produtos/1`, possui HATEOAS como os links mostrando a relação com as demais URI's presentes na API, e assim, essa API pode ser considerada RESTful.

Exemplo 1:

GET /produtos HTTP/1.1

HTTP/1.1 200 OK

```
[{
  "codigo": 1,
  "nome": "Samsung S10",
  "links": {
    "mostrarProduto": {
      "href": "/produtos/1"
    }
  }
},
{
  "codigo": 2,
  "nome": "Iphone 10",
  "links": {
    "mostrarProduto": {
      "href": "/produtos/2"
    }
  }
}]
```

Exemplo 2:

GET /produtos/1 HTTP/1.1

HTTP/1.1 200 OK

```
{
  "codigo": 1,
  "nome": "Samsung S10",
  "links": {
    "mostrarListaProdutos": {
      "href": "/produtos"
    }
  }
}
```

4.2.Criando uma API REST com Spring Boot

4.2.1.Criando conexão com banco de dados

Voltando ao projeto que já foi criado no capítulo 3 deste livro, é preciso primeiramente criar uma conexão com o banco de dados local para salvar e buscar os recursos da API e o banco de dados utilizado neste exemplo será o PostgreSQL. Mas antes de fazer a conexão, é preciso inserir mais duas dependências no arquivo pom.xml, Spring Data e a dependência do Postgres, como mostrado no exemplo abaixo.

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

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.2.2</version>
</dependency>
```

Figura 24: Dependência do Spring Data e JPA

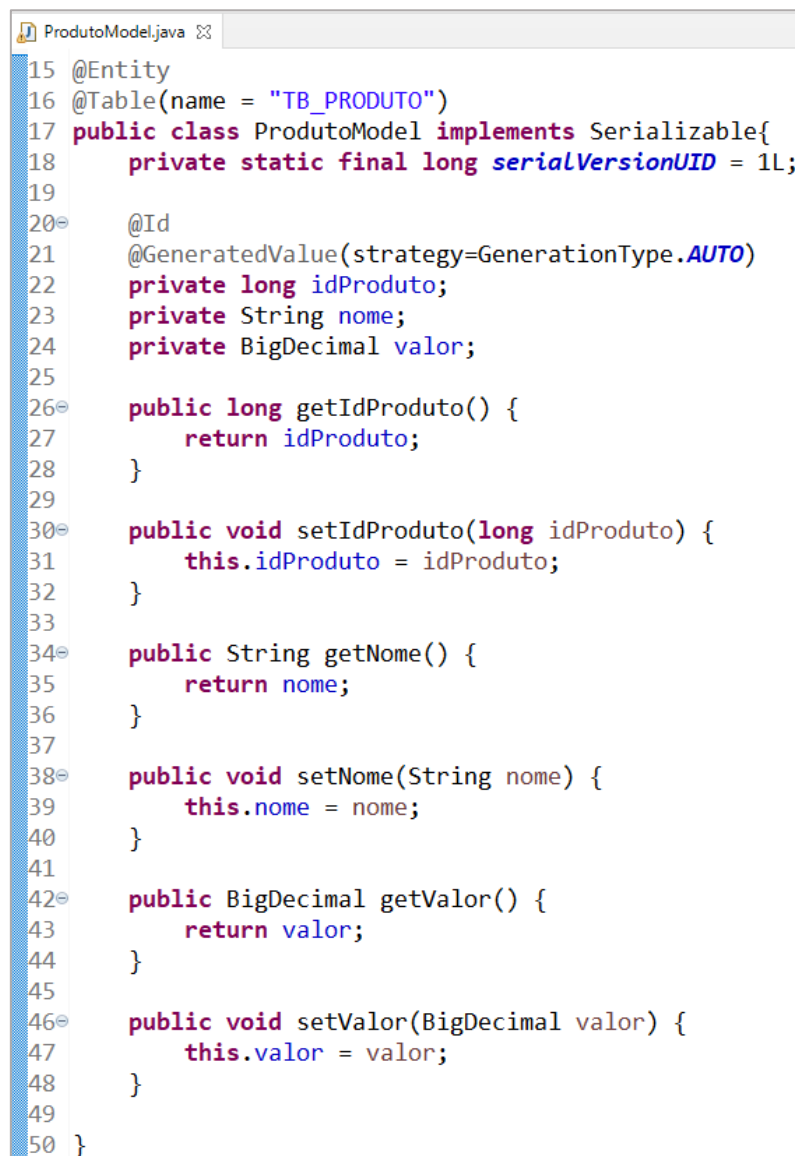
Após inserir essas dependências no projeto, é preciso configurar o banco de dados no arquivo application.properties, definindo url de acesso ao banco, nome do banco e suas credenciais, como mostrado no código abaixo.

```
application.properties
1 #Banco local - Michelli
2 spring.datasource.url= jdbc:postgresql://localhost:5432/apirest-springboot
3 spring.datasource.username=postgres
4 spring.datasource.password=banco123
5 spring.jpa.hibernate.ddl-auto=update
```

Figura 25: Configuração do banco de dados

4.2.2.Criando o Model e o Repository

Depois é preciso criar um model, objeto esse que será a representação do recurso na URI da API e esse model será um Produto que terá como atributos idProduto, nome e valor. Para isso, dentro do diretório raiz é preciso criar um novo diretório chamado models e dentro dele, criar uma nova classe ProdutoModel.class e anotá-lo com @Entity para que seja uma entidade no banco e @Table para definir o nome da tabela. É preciso também anotar o idProduto com @Id e @GeneratedValue(strategy=GenerationType.AUTO) para que o id seja gerado de forma automática em cada persistência de Produto no banco.



```
15 @Entity
16 @Table(name = "TB_PRODUTO")
17 public class ProdutoModel implements Serializable{
18     private static final long serialVersionUID = 1L;
19
20     @Id
21     @GeneratedValue(strategy=GenerationType.AUTO)
22     private long idProduto;
23     private String nome;
24     private BigDecimal valor;
25
26     public long getIdProduto() {
27         return idProduto;
28     }
29
30     public void setIdProduto(long idProduto) {
31         this.idProduto = idProduto;
32     }
33
34     public String getNome() {
35         return nome;
36     }
37
38     public void setNome(String nome) {
39         this.nome = nome;
40     }
41
42     public BigDecimal getValor() {
43         return valor;
44     }
45
46     public void setValor(BigDecimal valor) {
47         this.valor = valor;
48     }
49
50 }
```

Figura 26: Entidade Produto mapeada para o banco de dados

Como foi definida a configuração `spring.jpa.hibernate.ddl-auto=update` no `application.properties`, ao inicializar a aplicação com essa nova implementação, a tabela `tb_produto` deve ser criada no banco de forma automática, a qual pode ser visualizada na interface do pgAdmin do Postgres na imagem abaixo.

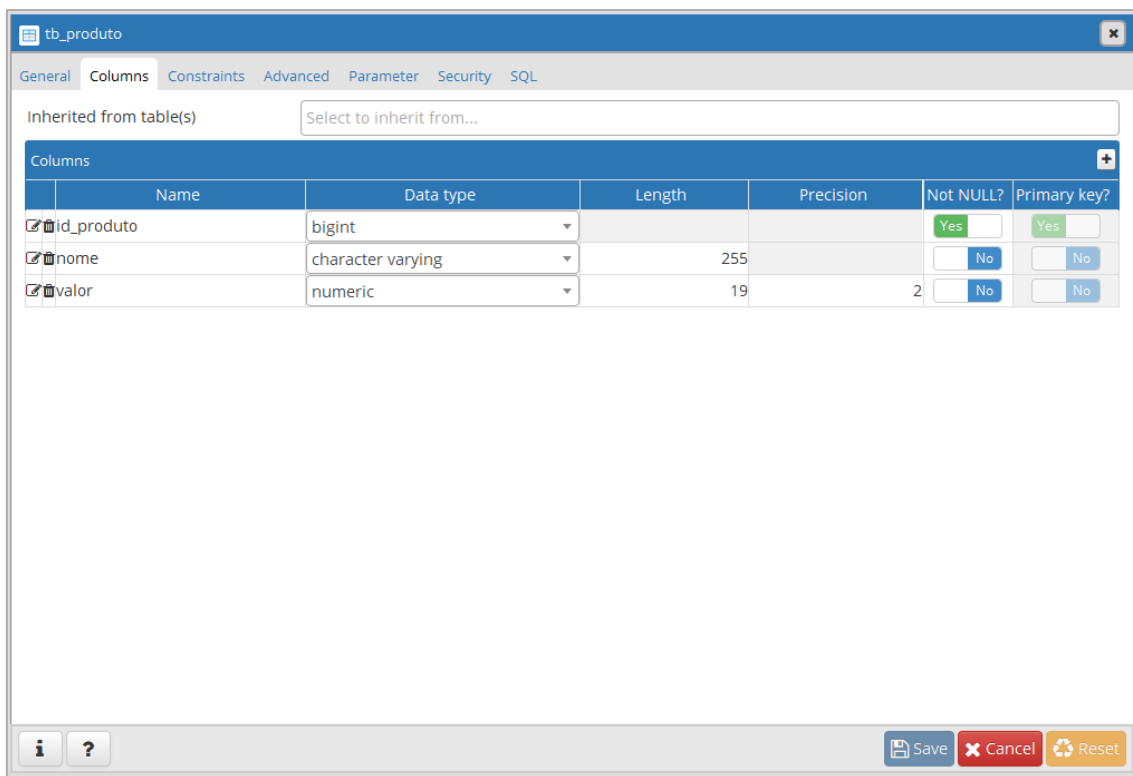
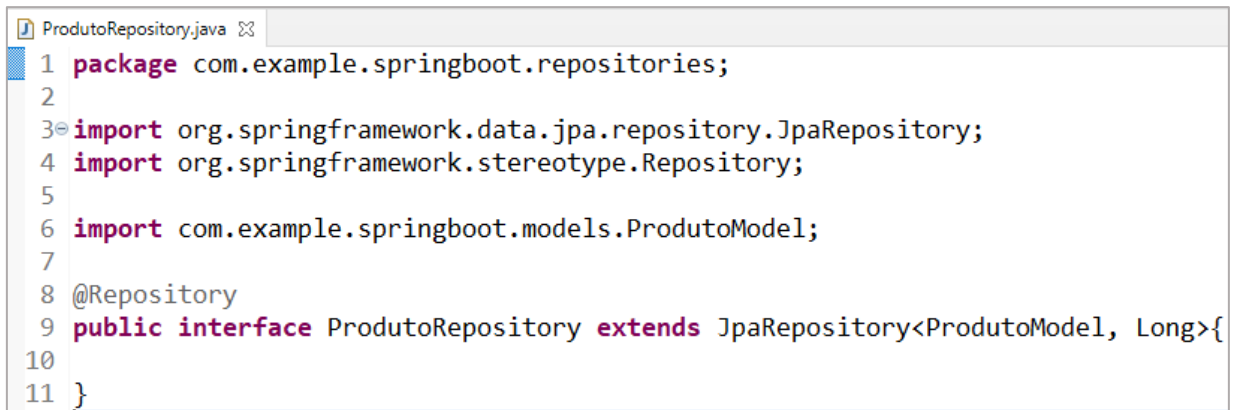


Figura 27: Interface pgAdmin mostrando os detalhes da `tb_produto`

Agora que já está pronto a conexão com o banco e a entidade criada, é preciso criar um novo diretório chamado `repositories` e dentro dele criar uma interface `ProdutoRepository` que estende `JpaRepository` do Spring Data para utilizar os métodos já prontos para transações com o banco de dados, como mostra o código abaixo.

A screenshot of an IDE window titled 'ProdutoRepository.java'. The code is as follows:

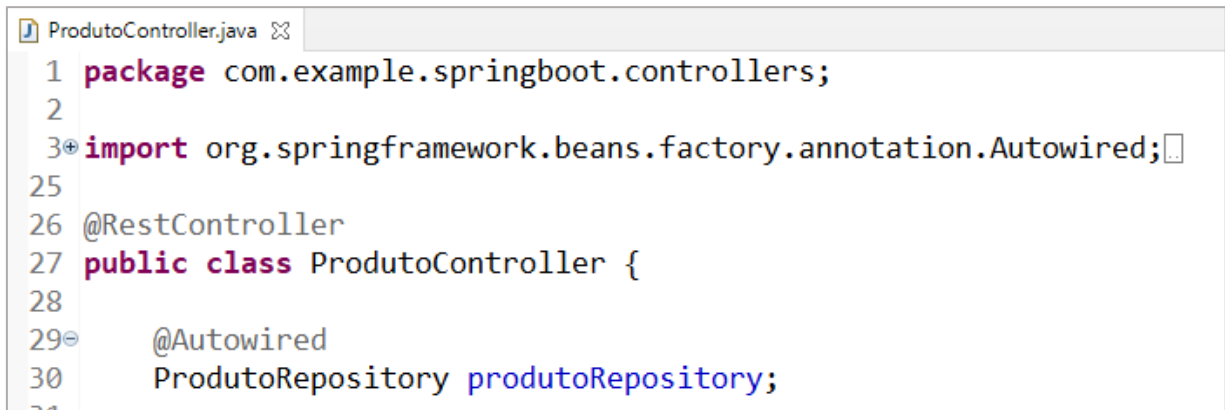
```
1 package com.example.springboot.repositories;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import org.springframework.stereotype.Repository;
5
6 import com.example.springboot.models.ProdutoModel;
7
8 @Repository
9 public interface ProdutoRepository extends JpaRepository<ProdutoModel, Long>{
10
11 }
```

Figura 28: Interface ProdutoRepository

Assim, métodos como `findAll()`, `findById(Long id)`, `save(S entity)`, `delete(S entity)`, entre outros podem ser utilizados sem a necessidade da implementação do zero. A interface foi anotada com `@Repository` para mostrar ao Spring que essa será um bean gerenciado por ele e como tal interface tem como objetivo transações com o banco de dados o melhor estereótipo a ser utilizado neste caso é `@Repository`.

4.2.3.Criando o Controller

O próximo passo para construção da API REST é criar o controller, onde será implementado os endpoints da aplicação. Em um novo diretório controllers é preciso criar uma nova classe `ProdutoController` e anotá-la com `@RestController`. Essa anotação é um estereótipo específico para criar endpoints REST e mostra ao Spring que tal classe será um bean gerenciado por ele através da IoC/ID. Como já foi criado o `ProdutoRepository` que também é um bean, é preciso criar um ponto de injeção dentro do controller para que o Spring injete as dependências de `ProdutoRepository` quando necessário.



```
1 package com.example.springboot.controllers;
2
3 import org.springframework.beans.factory.annotation.Autowired;
25
26 @RestController
27 public class ProdutoController {
28
29     @Autowired
30     ProdutoRepository produtoRepository;
```

Figura 29: Classe ProdutoController

4.2.4.Implementando os métodos GET ALL e GET ONE

Agora com o Controller pronto, é possível começar a implementar os métodos da API, iniciando pelos endpoints que retornam a listagem de produtos e um determinado produto passando seu id. Lembrando que é preciso definir bem o recurso, utilizar substantivos e definir os possíveis retornos utilizando o ResposnityEntity e o HttpStatus para montar a resposta completa com os códigos de retorno e os recursos esperados pelo cliente, e por fim utilizar de forma semântica o método HTTP, neste caso o método GET.

O código abaixo mostra os detalhes da implementação dos métodos HTTP GET que retornam primeiramente através da URI /produtos a listagem de produtos e através da URI /produtos/{id}, um produto em específico, caso este exista no banco de dados.

```

@GetMapping("/produtos")
public ResponseEntity<List<ProdutoModel>> getAllProdutos(){
    List<ProdutoModel> produtosList = produtoRepository.findAll();
    if(produtosList.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        return new ResponseEntity<List<ProdutoModel>>(produtosList, HttpStatus.OK);
    }
}

@GetMapping("/produtos/{id}")
public ResponseEntity<ProdutoModel> getOneProduto(@PathVariable(value="id") long id){
    Optional<ProdutoModel> produtoO = produtoRepository.findById(id);
    if(!produtoO.isPresent()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        return new ResponseEntity<ProdutoModel>(produtoO.get(), HttpStatus.OK);
    }
}

```

Figura 30: Implementação dos métodos GET ALL e GET ONE

Para buscar a listagem de produtos, utiliza-se o método `findAll()` que retorna `List<ProdutoModel>` e para buscar um produto específico através do id, utiliza-se o método `findById(Long id)` passando o id como argumento do método e obtendo como retorno um `Optional<ProdutoModel>`.

A fim de testar esses métodos criados, foi inserido no banco via comandos SQL alguns produtos e foram realizados os testes para esses dois endpoints utilizando o Postman, obtendo o seguinte resultado abaixo.

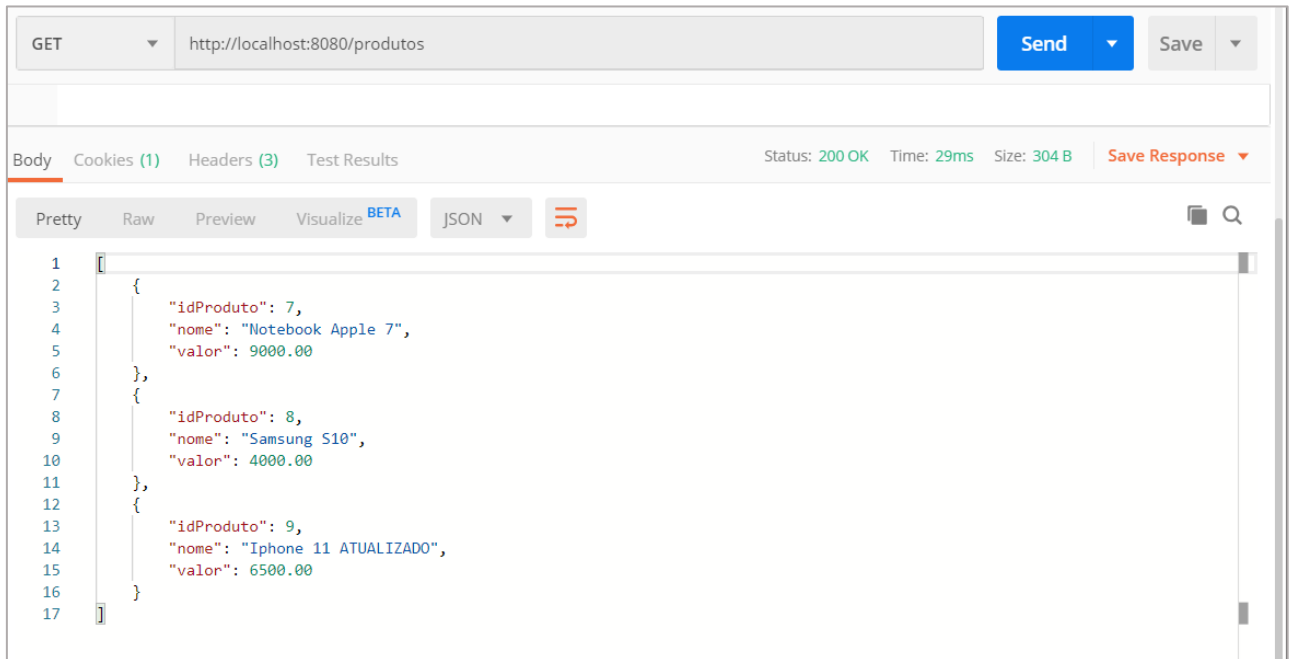
/produtos HTTP GET

Figura 31: Teste no Postman do método GET ALL

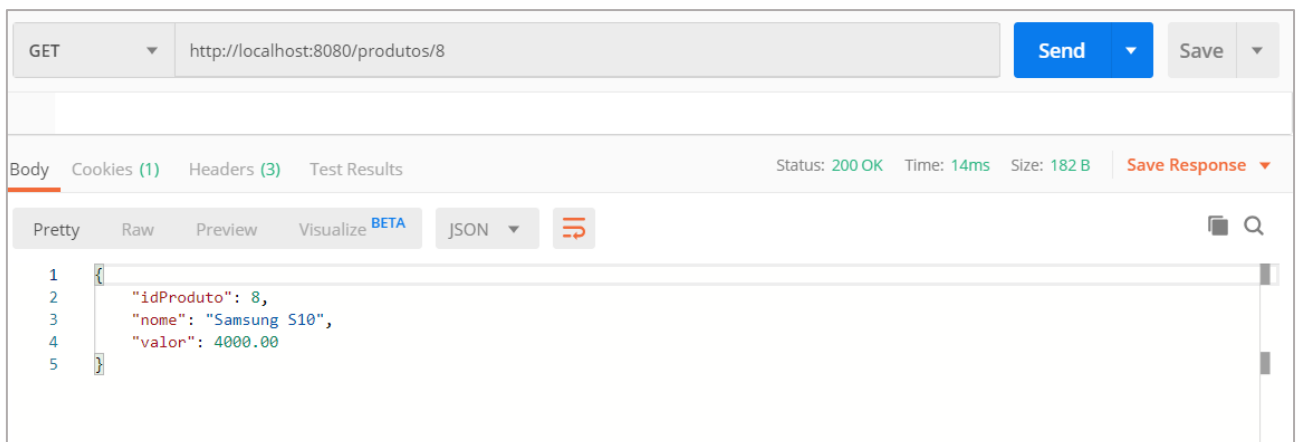
/produtos/8 HTTP GET

Figura 32: Teste no Postman do método GET ONE

4.2.5.Implementando os métodos POST, DELETE e PUT

O próximo passo é implementar os métodos POST, DELETE e PUT. Para salvar um novo produto através do método HTTP POST, utiliza-se o método `save(entity)` do JPA e caso a persistência aconteça corretamente no banco, o retorno deve ser `CREATED(201)`.

Para deletar um produto através do método HTTP DELETE, é preciso passar o id deste produto apenas e utilizar o método delete do JPA delete(S entity). O retorno caso a deleção ocorra corretamente deve ser OK(200).

E por fim, para atualizar um determinado produto é preciso passar o id deste produto para que seja inicialmente feito uma busca por ele no banco e caso exista, setar o id já existente e assim utilizar o método save(S entity) para salvar o produto com o mesmo id já existente, ou seja, atualizá-lo no banco de dados.

Todo esse passo a passo descrito acima pode ser visualizado no código abaixo.

```
@PostMapping("/produtos")
public ResponseEntity<ProdutoModel> saveProduto(@RequestBody @Valid ProdutoModel produto) {
    return new ResponseEntity<ProdutoModel>(produtoRepository.save(produto), HttpStatus.CREATED);
}

@DeleteMapping("/produtos/{id}")
public ResponseEntity<?> deleteProduto(@PathVariable(value="id") long id) {
    Optional<ProdutoModel> produtoO = produtoRepository.findById(id);
    if(!produtoO.isPresent()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        produtoRepository.delete(produtoO.get());
        return new ResponseEntity<>(HttpStatus.OK);
    }
}

@PutMapping("/produtos/{id}")
public ResponseEntity<ProdutoModel> updateProduto(@PathVariable(value="id") long id,
    @RequestBody @Valid ProdutoModel produto) {
    Optional<ProdutoModel> produtoO = produtoRepository.findById(id);
    if(!produtoO.isPresent()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        produto.setId(produtoO.get().getId());
        return new ResponseEntity<ProdutoModel>(produtoRepository.save(produto), HttpStatus.OK);
    }
}
```

Figura 33: Implementação dos métodos POST, DELETE e PUT

Realizando os testes dessas ultimas implementações no Postman, os retornos para as requisições aos métodos POST, PUT e DELETE de produtos podem ser visualizados nas imagens abaixo.

/produtos HTTP POST

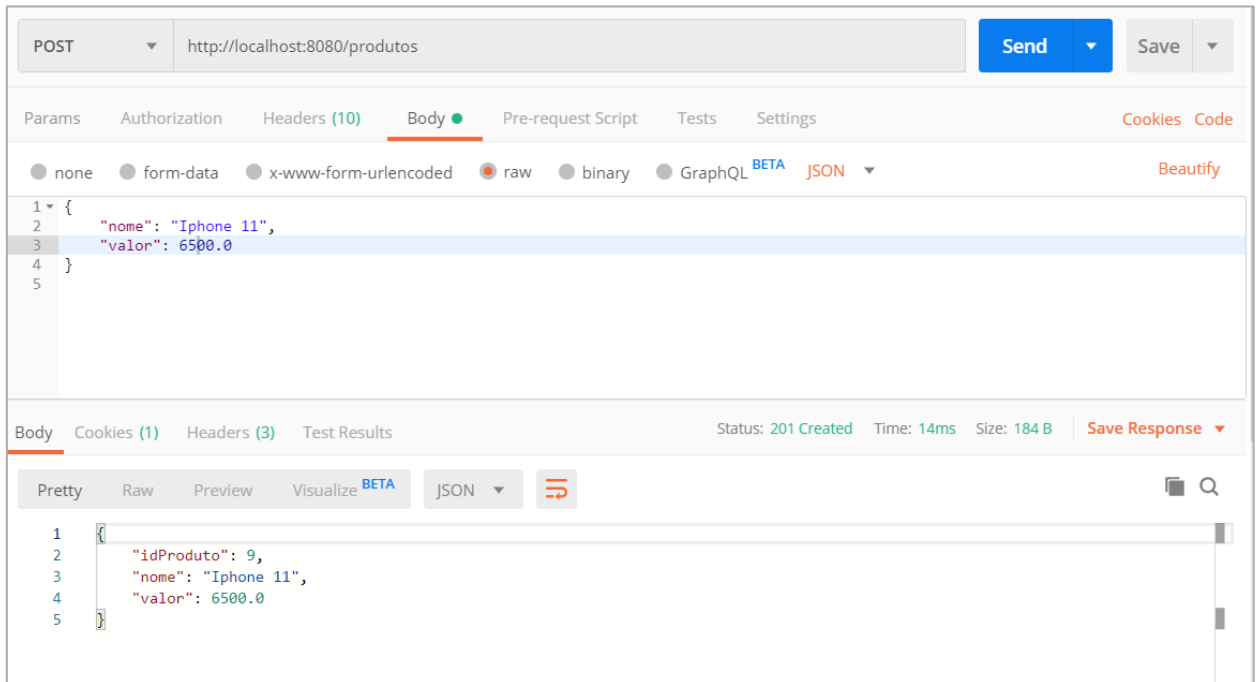


Figura 34: Teste no Postman do método POST

/produtos/9 HTTP PUT

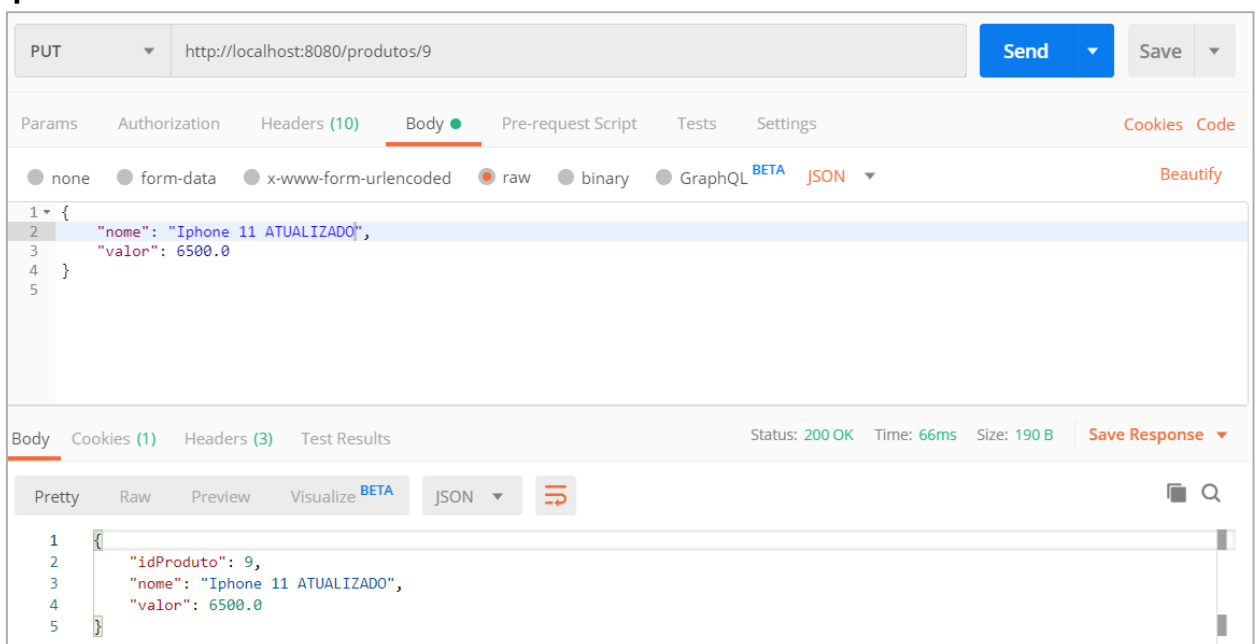


Figura 35: Teste no Postman do método PUT

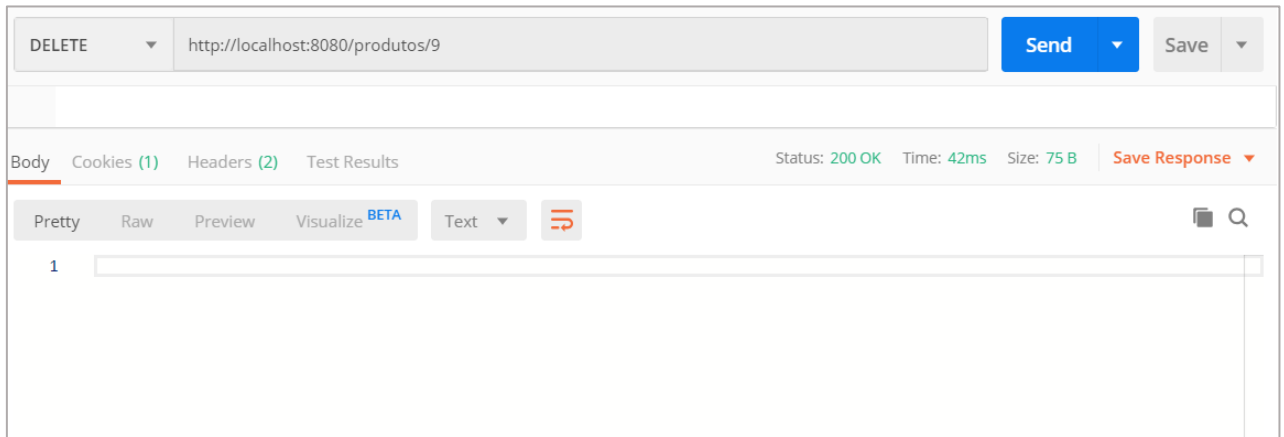
/produtos/9 HTTP DELETE

Figura 36: Teste no Postman do método DELETE

Agora para que essa API de Produtos possa ser considerada uma API RESTful e atingir o nível 3 da maturidade de Richardson, é preciso inserir as hipermídias (HATEOAS).

4.2.6. Inserindo as HATEOAS

Para implementar as HATEOAS na API é preciso inserir mais uma dependência no arquivo pom.xml do projeto, como mostra o código abaixo.

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

Figura 37: Dependências das HATEOAS

De acordo com o modelo de Maturidade de Richardson, uma API deve possuir as hipermídias que mostram o seu estado atual e seu relacionamento com os elementos ou estados futuros para que seja então considerada RESTful. Assim, para que essa API atinja este nível, o recurso Produto deve apresentar os seus atributos e também os links, onde será mostrado o caminho de relacionamento com os demais elementos.

Para isso, na entidade `ProdutoModel` é preciso estender `RepresentationModel` para que através do seu método `add()` a classe `ProdutoModel` exiba o link das demais URI's relacionadas, como mostra o código abaixo.



```

15 @Entity
16 @Table(name = "TB_PRODUTO")
17 public class ProdutoModel extends RepresentationModel<ProdutoModel> implements Serializable{
18     private static final long serialVersionUID = 1L;
19
20     @Id
21     @GeneratedValue(strategy=GenerationType.AUTO)
22     private long idProduto;
23     private String nome;
24     private BigDecimal valor;
25
26     public long getIdProduto() {
27         return idProduto;
28     }
29
30     public void setIdProduto(long idProduto) {
31         this.idProduto = idProduto;
32     }
33
34     public String getNome() {
35         return nome;
36     }
37
38     public void setNome(String nome) {

```

Figura 38: `ProdutoModel` extends `ResourceSupport`

Quando uma requisição for solicitada para retornar uma lista de produtos ou um determinado produto é preciso definir o link que será adicionado em cada caso e construí-lo dentro dos métodos do controller e então, adicioná-lo a cada produto.

Para isso, pode ser utilizado o método `linkTo()`, o qual irá construir uma URI de acordo com o controller e o método definido, `methodOn()` faz o mapeamento do método de destino da chamada e `withSelfRel()` e `withRel()` criam um auto link de acordo com a relação.

Cada link criado deve ser inserido no produto através do método `add()`, como pode ser observado no código abaixo.

```

@GetMapping("/produtos")
public ResponseEntity<List<ProdutoModel>> getAllProdutos(){
    List<ProdutoModel> produtosList = produtoRepository.findAll();
    if(produtosList.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        for(ProdutoModel produto : produtosList) {
            long id = produto.getIdProduto();
            produto.add(LinkTo(methodOn(ProdutoController.class).getOneProduto(id)).withSelfRel());
        }
        return new ResponseEntity<List<ProdutoModel>>(produtosList, HttpStatus.OK);
    }
}

```

Figura 39: Criando e adicionando o link para a listagem de produtos.

```

@GetMapping("/produtos/{id}")
public ResponseEntity<ProdutoModel> getOneProduto(@PathVariable(value="id") long id){
    Optional<ProdutoModel> produto0 = produtoRepository.findById(id);
    if(!produto0.isPresent()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
        produto0.get().add(LinkTo(methodOn(ProdutoController.class).getAllProdutos()).withRel("Lista de Produtos"));
        return new ResponseEntity<ProdutoModel>(produto0.get(), HttpStatus.OK);
    }
}

```

Figura 40: Criando e adicionando o link para um único produto.

Ao enviar uma requisição ao servidor solicitando a listagem de produtos, agora além dos atributos de `ProdutoModel`, também será retornado o link que mostra o caminho e relacionamento para acessar cada produto individualmente, através da url `http://localhost:8080/produtos/{id}`.

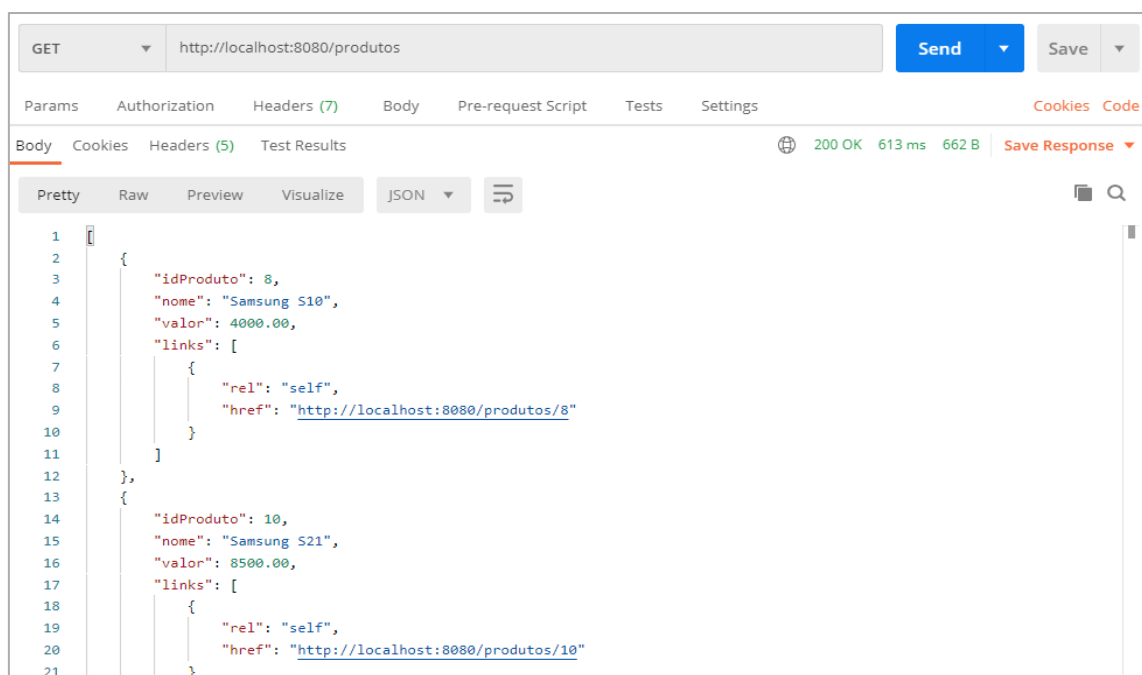


Figura 41: Teste no Postman do método GET ALL com HATEOAS

E da mesma maneira, ao enviar uma requisição ao servidor solicitando um determinado produto pelo id, será retornado o link mostrando o caminho completo para o acesso e retorno a listagem de produtos <http://localhost:8080/produtos>, como pode ser visualizado na imagem abaixo.

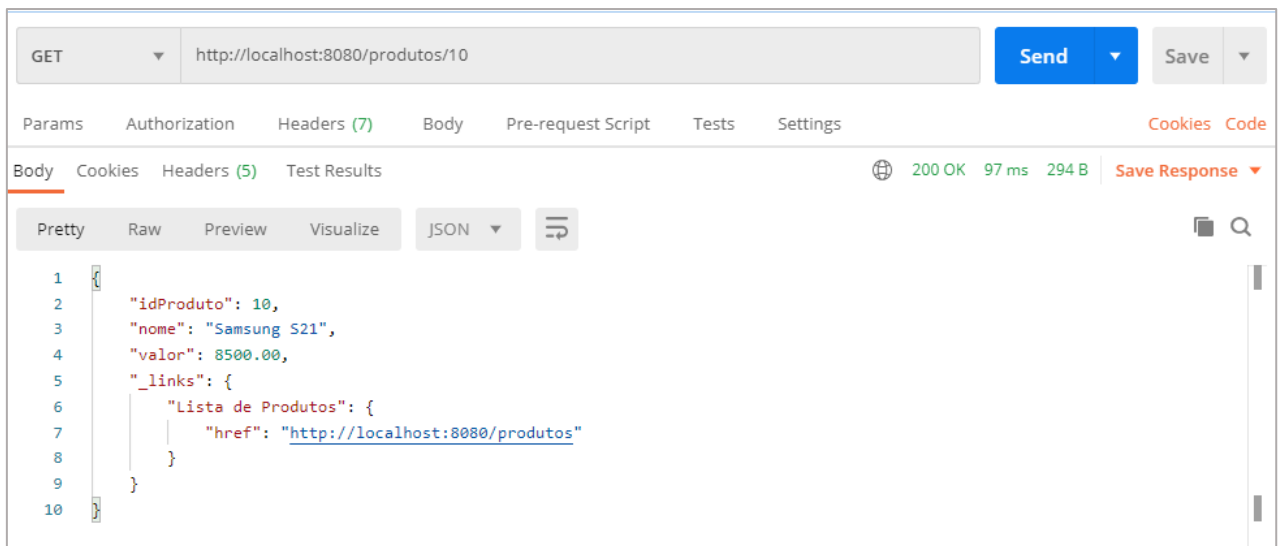


Figura 42: Teste no Postman do método GET ONE com HATEOAS

Agora sim, com a inserção das HATEOAS na aplicação e feita as configurações necessárias, pode-se dizer que a API de produtos atingiu o nível 3 de maturidade e pode ser considerada uma API RESTful.

5. *Microservices* com Spring Boot

5.1. Arquitetura de Microservices

Cada vez mais as empresas estão aderindo a arquitetura de microservices como uma nova opção para substituir a arquitetura monolítica, a qual se baseia em vários módulos em um único sistema, ou seja, se define por ser um grande bloco de códigos onde estão presentes todas as regras de negócio e funcionalidades da aplicação. Geralmente uma aplicação que foi construída em cima da arquitetura monolítica costuma ter uma única base de dados utilizada e acessada por todo o sistema.

Como um exemplo de aplicação com arquitetura monolítica, imagine um sistema ERP, o qual tem como responsabilidades o controle de compra, venda, finanças, contabilidade, entre outras operações dentro de uma corporação. Cada uma dessas funcionalidades se baseia em um módulo, porém todos esses módulos compõem uma única aplicação, sendo fortemente acoplados e acessando a mesma base de dados, como pode ser observado na imagem abaixo.

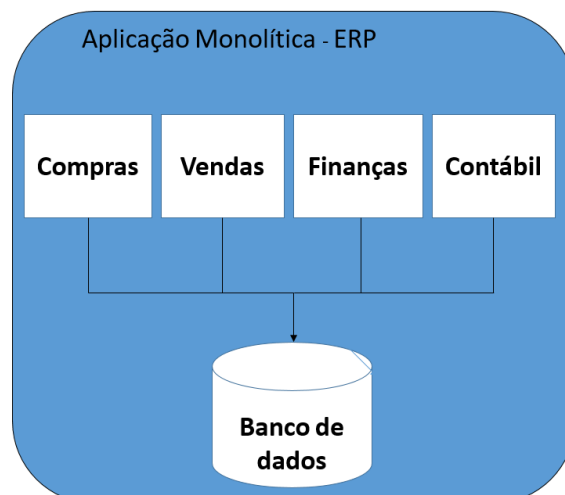


Figura 43: Exemplo arquitetura monolítica

Já a arquitetura de microservices é baseada na construção de uma aplicação dividida em vários serviços menores, bem específicos e objetivos sendo também independentes entre si. Então voltando ao exemplo do sistema ERP, se considerar agora que esse mesmo sistema foi construído utilizando a arquitetura de microservices, cada uma das funcionalidades dessa aplicação seria um micro serviço independente e específico de acordo com sua responsabilidade dentro do sistema, cada qual utilizando sua própria base de dados, como pode ser observado na imagem abaixo.

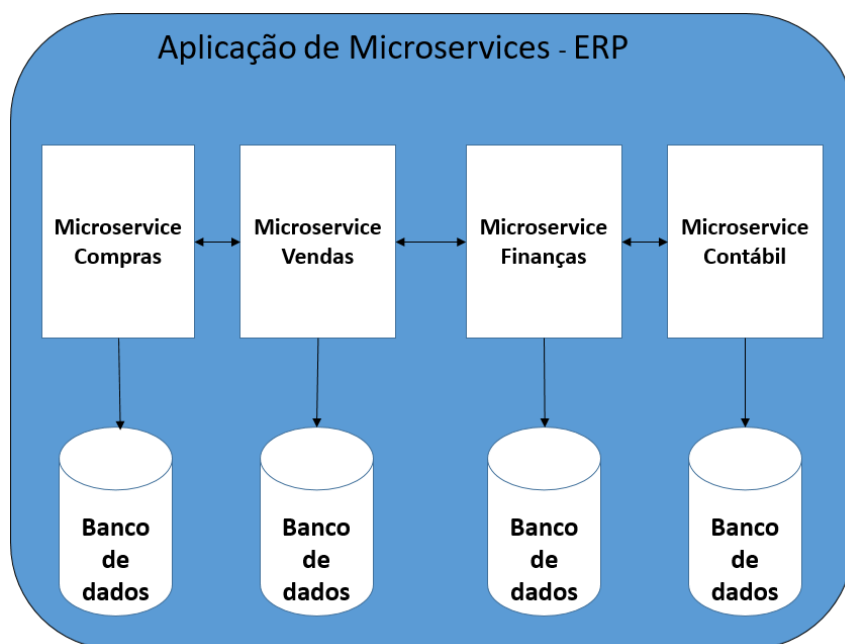


Figura 44: Exemplo arquitetura de microservices

Quando é preciso fazer uma manutenção no sistema ou uma nova implementação, seja corrigir um bug em produção ou inserir uma nova funcionalidade, utilizando uma arquitetura de microservices esse processo fica muito mais simples, pois não há a necessidade de parar todo o sistema e sim, apenas o micro serviço em questão para fazer a correção ou implementação. O serviço que ficará indisponível não afetará o funcionamento dos demais e assim, os outros serviços que possuem funcionalidades diferentes poderão continuar respondendo para seus clientes normalmente.

Assim a disponibilidade da aplicação fica muito maior quando comparada com um sistema construído no modo monolítico, o qual para realizar o mesmo processo seria necessário parar a aplicação como um todo.

Escalar uma aplicação é um processo bem mais flexível e customizável quando se utiliza microservices, pois é possível escalar apenas os serviços mais acessados por exemplo, fazendo com que a aplicação apresente um bom controle de performance. Se tratando de um sistema monolítico, não há como fazer esse controle, é preciso escalar a aplicação como um todo já que não há como dividir os módulos que são mais acessados e menos acessados e assim, não há um bom controle de performance.

Outro ponto relevante é que na arquitetura de microservices, como cada micro serviço costuma ter sua própria base de dados, caso ocorra alguma alteração na modelagem de dados atual em algum dos serviços, tal modificação não afeta os demais. Isso não acontece na arquitetura monolítica, onde todos os módulos acessam a mesma base e assim ficam passíveis de erros ou quebra de regras caso ocorram alterações na modelagem de dados.

Outra vantagem da arquitetura em microservices é que cada micro serviço pode ser construído utilizando a linguagem de programação que for mais adequada para cada caso gerando um baixo acoplamento na aplicação, o que é o mais adequado pelos padrões de projeto. Como os serviços são independentes, um não precisa saber como o outro foi construído, apenas precisam se comunicar quando houver necessidade independente da linguagem. Quando se trata do estilo monolítico, o sistema como um todo precisa ser construído em cima da mesma linguagem, já que não é possível dividi-lo em módulos e assim a aplicação fica com um alto acoplamento.

Um ponto positivo da arquitetura monolítica é que para fazer o deploy da aplicação em uma cloud basta gerar um único arquivo war ou jar. Na arquitetura de microservices, o deploy a princípio pode parecer um pouco mais robusto, já que a aplicação como um todo conta com vários serviços cada qual com um deploy específico. Porém, as plataformas cloud fornecem várias facilidades para esse processo, como por exemplo, o uso de pipelines que depois de

configuradas geram deploys automáticos a cada push no git em um determinado ambiente. Assim, depois de configurada uma vez, o deploy desses microservices passa a ser algo mais simples e de fácil controle, porém é preciso seguir as boas práticas e ferramentas que as plataformas cloud oferecem para facilitar e controlar esses processos.

5.2. Comunicação entre Microservices

Na arquitetura de microservices os serviços são independentes entre si, porém na maioria das vezes há a necessidade da comunicação entre eles. Dentre as formas mais utilizadas para essa comunicação estão a comunicação através do protocolo HTTP utilizando de API's REST por exemplo, e a comunicação através do protocolo AMQP, utilizando de mensagerias.

O tipo de comunicação entre os microservices a ser utilizado nesse tipo de arquitetura deve ser definido de acordo com o funcionamento do sistema e das principais necessidades. No caso de sistemas síncronos pode ser utilizado a comunicação pelo protocolo HTTP criando uma API para comunicação utilizando os padrões JSON ou XML, e neste caso, a comunicação é feita entre dois serviços, onde o requisitante necessita receber um retorno do serviço requisitado para completar seu processo.

Já em sistemas que funcionam de forma assíncrona a maneira mais apropriada de comunicação é por mensagerias através do protocolo AMQP, casos em que a comunicação é feita para a entrega de uma mensagem de um serviço para outro, não necessitando de retornos, apenas a garantia de entrega de tal mensagem.

5.3.Exemplo de Microservices com Spring Boot

Como o processo de iniciar uma aplicação com Spring Boot se tornou algo bem simples e rápido, as empresas optam por utilizar deste framework para construir essa arquitetura de microservices, onde além do Spring Boot trazer essa elevada produtividade a princípio, a plataforma Spring já conta com vários projetos prontos para serem utilizados em sistemas distribuídos, permitindo com que aplicações desde as mais simples até as mais complexas sejam construídas com base nesse framework.

O projeto Spring Cloud possui vários módulos específicos para diversas funcionalidades em um sistema distribuído. Dentre eles, o Spring Cloud Gateway para criação de um gateway da aplicação, Spring Cloud Netflix Eureka ou Spring Cloud Consul que são serviços de descoberta de microservices na arquitetura e o Spring Cloud Stream, que fornece suporte para mensageria como por exemplo RabbitMQ e Apache Kafka. A lista completa de todos os módulos do Spring Cloud pode ser visualizada neste link <https://spring.io/projects/spring-cloud>.

Como exemplo prático, vamos considerar uma aplicação de controle de produtos que disponibiliza de API's para que uma aplicação front-end com Angular possa construir a interface de um sistema que mostra a lista de produtos de uma determinada loja online e também, qual a unidade que possui estoque de tal produto a pronta entrega.

Para isso, primeiramente é preciso construir no back-end uma aplicação que retorne essas API's de produtos com os detalhes do estoque. Pensando em sistema distribuído, a princípio já é possível dividir essa aplicação em três serviços específicos e independentes, um para controle de produtos, outro para controle de estoque e um para centralização das entradas na aplicação e autenticação (Gateway).

Neste caso, quando o serviço de controle de produtos recebe uma requisição para retornar ao cliente (aplicação angular), seja a listagem de produtos ou os detalhes de um produto específico, ele precisa consultar os dados de estoque que são de responsabilidade do serviço controle de estoque, já que como são

serviços diferentes, cada um possui sua própria base de dados. Assim, analisando o funcionamento dessa aplicação em termos de comunicação, o serviço controle de produtos envia internamente uma requisição para o serviço controle de estoque esperando um retorno deste com os dados necessários. Neste caso, como o sistema se caracteriza síncrono o tipo de comunicação mais apropriado seria utilizando o protocolo HTTP, onde através de uma requisição GET via HTTP REST o serviço controle de produtos envia ao serviço controle de estoque uma requisição pedindo os detalhes de estoque para esse produto. Tal fluxo pode ser visualizado na imagem abaixo.

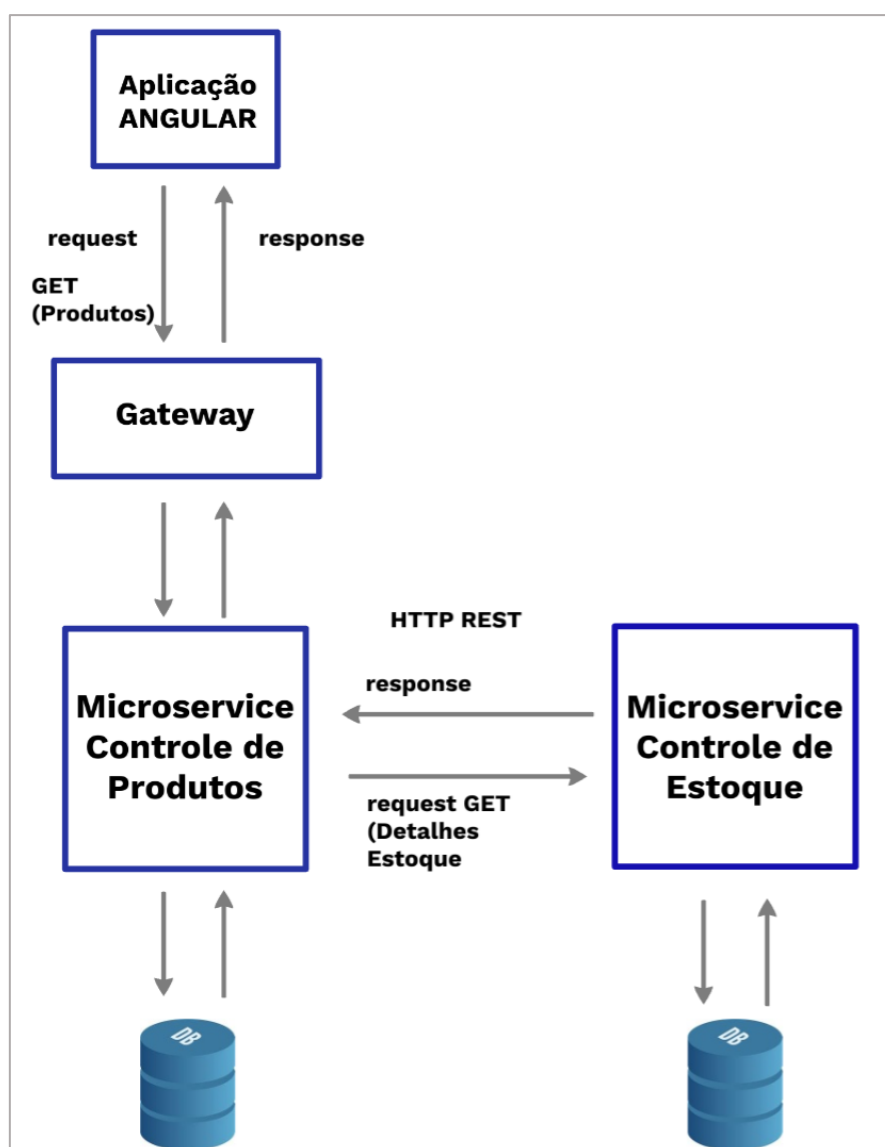


Figura 45: Fluxo da comunicação entre microservices.

Assim, quando o serviço controle de estoque retornar a informação ao controle de produtos, este irá finalizar seu processamento, completar a resposta e retornar ao cliente a listagem de produtos com os detalhes de estoque.

Cada um desses microservices podem ser construídos como sendo uma aplicação Spring Boot e esses serviços, controle de produtos e controle de estoque, serão nada mais nada menos que uma API REST similar à que foi construída no capítulo 4 deste livro, que disponibiliza endpoints de seus recursos. Assim, tanto a aplicação Angular quanto os próprios serviços internos poderão enviar e receber requisições utilizando o protocolo HTTP.

O Spring Cloud pode ser utilizado para fazer o controle gateway com Spring Cloud Gateway por exemplo, utilizar o Spring Netflix Eureka ou Spring Cloud Consul para descobertas de serviços e Spring Cloud Config para configurações. Utilizar também o Spring MVC já que cada um desses microservices (controle de estoque e produtos) serão aplicações web e o Spring Data para comunicação/transação com banco de dados, entre outros módulos da plataforma Spring.

Agradecimentos

Se você chegou até aqui significa que agora já tem uma boa base dos principais conceitos e tecnologias para se tornar um desenvolvedor Java Web utilizando Spring para construir API's REST e microservices com Spring Boot.

É sempre importante conciliar a teoria aprendida com exemplos práticos, então aplicar os conceitos na prática sempre irá reforçar o aprendizado e melhorar o entendimento. É essencial que os estudos continuem, afinal a plataforma Spring possui muitos projetos e módulos diferentes que podem ser estudados e utilizados na construção das aplicações.

Para quem tiver dúvidas, sugestões ou críticas podem entrar em contato por meio dos acessos ou redes sociais apresentadas no começo do livro para que eu sempre possa estar melhorando e trazendo um conteúdo mais rico e assertivo nas próximas edições.

Obrigada!

