JAVA COM SPRING BOOT Parte 01

O Spring Boot é um **framework** Java que facilita muito a criação de aplicações web e serviços backend.

FRAMEWORK

Um framework é como uma base pronta para você construir um projeto de software.

Em vez de começar tudo do zero, o framework já traz várias ferramentas, bibliotecas e regras que te ajudam a criar o que você precisa, mais rápido e organizado.

De maneira simples:

Ele já vem com códigos prontos e estruturas.

Ele define um "jeito de fazer" as coisas (por exemplo, onde colocar arquivos, como tratar erros, como criar telas, etc.).

Você escreve o seu código seguindo esse modelo que ele oferece.

Exemplos:

Spring Boot → para criar aplicações em Java.
Django → para criar sites e APIs em Python.
React → para criar interfaces de sites em JavaScript.
Laravel → para sites em PHP.

Analogia rápida:
Se programar fosse construir uma casa, usar um framework seria como usar kits de construção e projetos prontos, em vez de ter que fazer tudo na mão (como fabricar cada tijolo!).

Continuado com Spring Boot

Ele é uma ferramenta que ajuda você a criar aplicações em Java de maneira rápida, simples e com pouca configuração manual.

Faz parte do ecossistema Spring, mas com foco em tornar tudo mais "automático" — por exemplo, ele configura sozinho muitas coisas que você teria que configurar manualmente em um projeto Java tradicional.

Principais características do Spring Boot:

Criação fácil de **APIs RESTful.**

API RESTFull

API RESTful é um jeito de criar sistemas que se comunicam pela internet usando regras simples e padrões.

Como funciona?
Tudo é um recurso (usuários, produtos, etc.) e cada um tem um endereço único (URL).

Ex.: https://api.com/usuarios

Usa ações básicas do HTTP:

GET → Buscar dados.--> GET é tudo que vem pela barra de endereço do navegador.
POST → Criar algo novo.
PUT/PATCH → Atualizar.
DELETE → Remover.

Retorna dados em formato fácil (geralmente JSON).

Json

```
{
   "nome": "João",
   "idade": 30
}
```

Não guarda estado: Cada requisição é independente (como abrir um novo pedido no restaurante a cada vez).

Por que é bom?
✓ Fácil de entender e usar.
✓ Rápida e leve.
✓ Funciona em qualquer plataforma (web, app, IoT).

É o padrão mais usado hoje para conectar apps e serviços!

Configurações automáticas (auto-configuration).

Embute servidor como **Tomcat** — você não precisa instalar nada separado.

Tomcat

O Tomcat é um servidor web e contêiner de servlets desenvolvido pela Apache Software Foundation. Ele é usado principalmente para executar aplicações Java baseadas em Java Servlets, JavaServer Pages (JSP) e Java EE (agora Jakarta EE).

Para que serve o Tomcat?

Executa aplicações Java web → Hospeda sistemas feitos com Servlets, JSP e frameworks como Spring MVC, JSF, etc.

Funciona como um servidor web leve → Pode servir páginas HTML, arquivos estáticos e APIs RESTful.

Gerencia o ciclo de vida de Servlets/JSP → Controla a inicialização, execução e destruição de componentes Java web.

Suporta o protocolo HTTP/HTTPS → Permite comunicação com navegadores e clientes via web.

Quando usar o Tomcat?

Se você precisa rodar uma aplicação web simples (API REST, site em JSP).

Se está usando Spring Boot (que já vem com um Tomcat embutido).

Se quer um servidor fácil de instalar e configurar.

Resumindo: Tomcat é um servidor simples e eficiente para rodar aplicações Java web. Se você está começando em Java para web, ele é uma ótima escolha!

Sistema de dependências muito organizado (com o Maven ou Gradle).

MAVEN

Maven é uma das ferramentas mais populares para gerenciamento de dependências e automação de builds no mundo Java. Ele foi projetado para facilitar o processo de construção, documentação e gerenciamento de projetos em Java, além de fornecer uma maneira padronizada de gerenciar dependências externas, como bibliotecas e frameworks.

Principais características do Maven:
Gerenciamento de dependências:

Maven permite que você declare as dependências do seu projeto (bibliotecas externas que seu código precisa) no arquivo pom.xml (Project Object Model). Ele baixa automaticamente essas dependências de repositórios como o Maven Central.

Por exemplo, se o seu projeto precisa de uma biblioteca como o Spring Boot, você só precisa declarar essa dependência no pom.xml, e o Maven faz o download para você.

Fácil de fazer deploy (inclusive em nuvem).

> "Deploy" na programação (em inglês, "deployment") significa implantar, publicar ou colocar um sistema ou aplicação em funcionamento, geralmente em um ambiente de produção (como um servidor na internet).
> Em termos simples:
> Deploy é o processo de colocar um software no ar, para que os usuários possam acessá-lo e usá-lo.

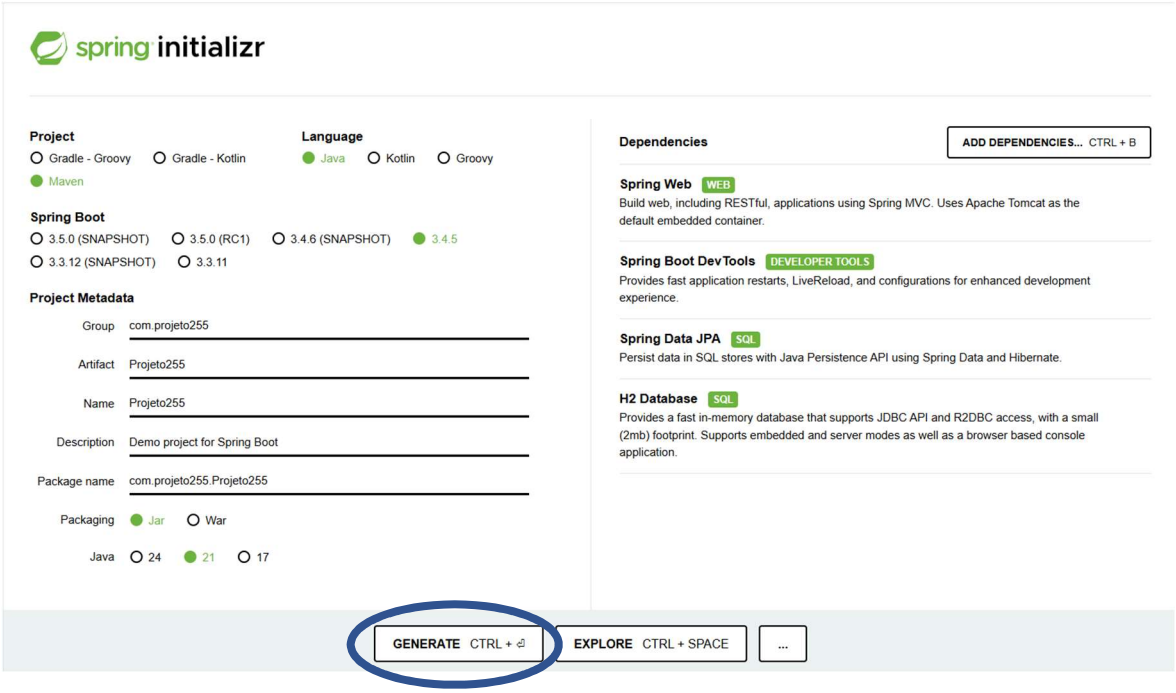
Exemplo de onde é usado:

Se você já usou ou viu um aplicativo que precisa de um servidor para processar pedidos, como um app de delivery ou um sistema de gestão, provavelmente algo como Spring Boot pode estar por trás do backend.

Resumo em uma frase:
Spring Boot é a maneira moderna e rápida de criar servidores e APIs em Java.

Vamos trabalhar

start.spring.io

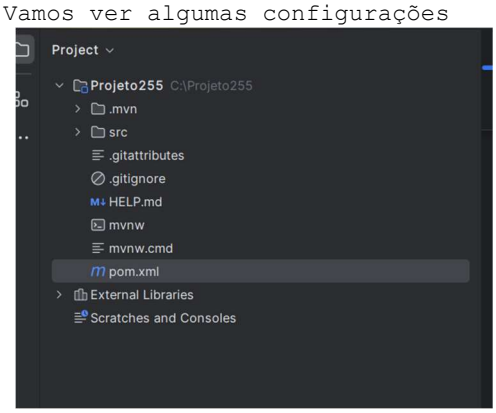


Depois deverá gerar um arquivo Projeto255.zip na pasta download.

Descompacte este arquivo e coloque por exemplo Disco Local C:

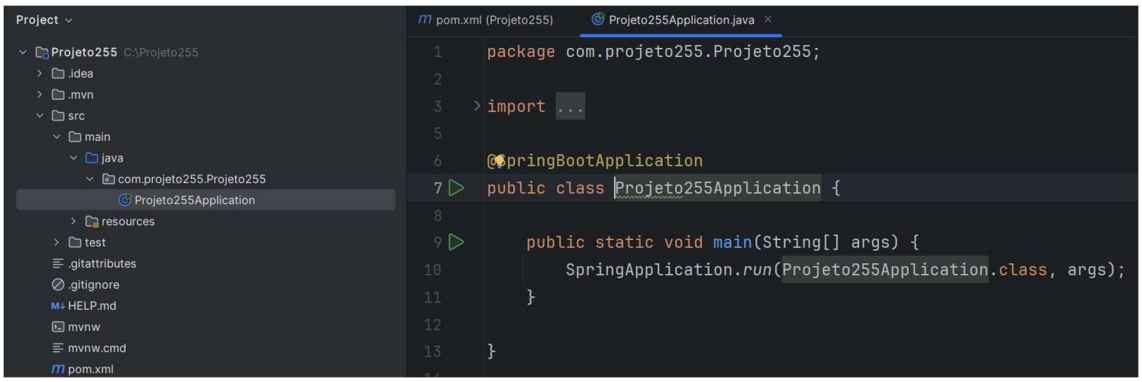
Vamos abrir esta pasta com o Intellij.

Vamos ver algumas configurações



Arquivo pom.xml

O arquivo pom.xml (Project Object Model) é fundamental em projetos que utilizam o Apache Maven, uma ferramenta de automação de builds (compilação, empacotamento, testes, etc.) em projetos Java.

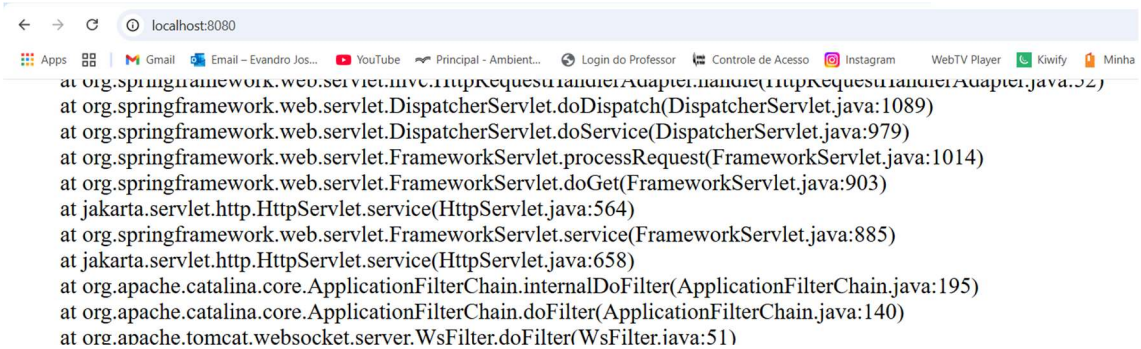


src → main → java → com.projeto.Projeto255 → Projeto255Application

É o nosso arquivo principal, vamos executa-los
Deve aparecer como a tela abaixo

Agora vamos ver o servidor Tomcat funcionando

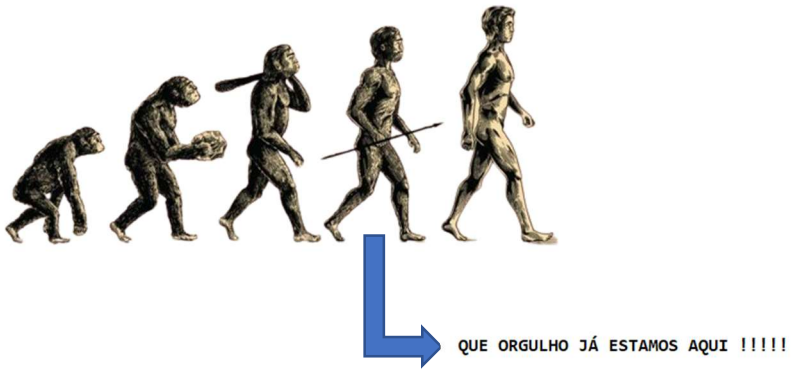
Abra um navegador e digite "localhost:8080"



Se der o erro acima e que ta certo!!!!

Nosso famoso "Ola Mundo" em Spring Boot

Vamos na classe principal

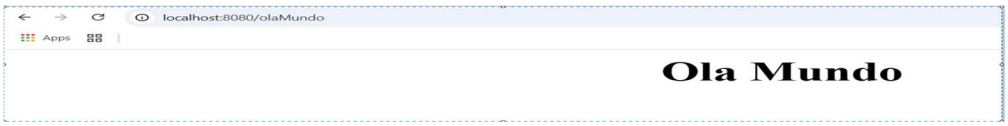


```java
package com.projeto255.Projeto255;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication

@RestController 3

public class Projeto255Application {

    @GetMapping("olaMundo") 2

        public String olaMundo(){
                return "<h1 align=center>Ola Mundo!!"; 1
        }


        public static void main(String[] args) {
                SpringApplication.run(Projeto255Application.class, args);
        }

}
```

1 – Aqui estamos criando apenas um método que retorna uma string.

2- @GetMapping no Spring Boot é uma Notação usada para mapear requisições HTTP GET para um método de um controller.("OlaMundo") significa se for digitado "OlaMundo" no navegador, será mapeado para o método abaixo.

3- @RestController é uma Notação do Spring que diz que a classe controller (que recebe trata requisições HTTP.



Bem agora vamos fazer uma coisa prática vamos cadastrar produtos, já estávamos mexendo com produtos anteriormente.

Volte ao estado original do nosso arquivo principal, ou seja, vamos retirar as anotações e método que colocamos.
MOSTRAR
CRTL + ALT + o (Organizer).

```java
package com.projeto255.Projeto255;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Projeto255Application {

        public static void main(String[] args) {
                SpringApplication.run(Projeto255Application.class, args);
        }
}
```

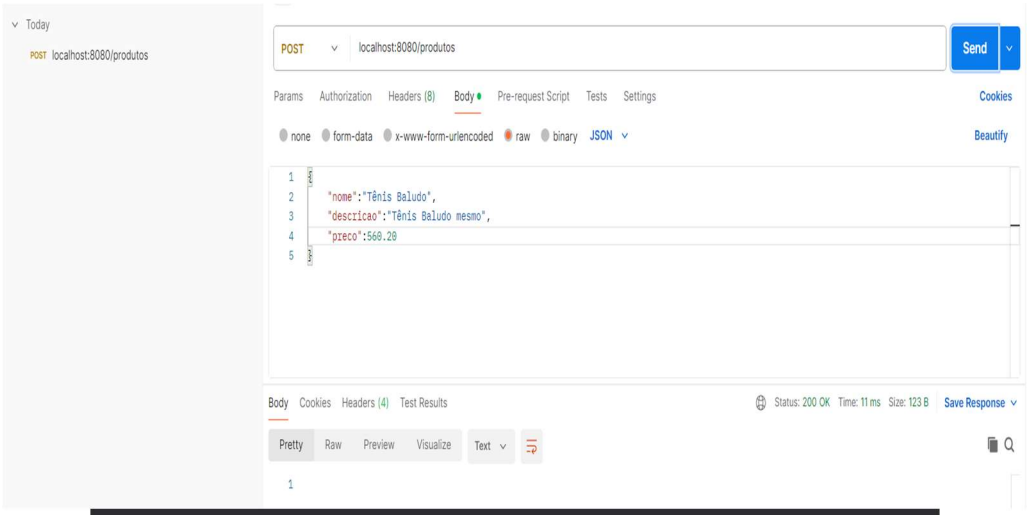
Vamos criar a nossa classe Produto, mas vamos coloca-la dentro de um pacote chamado Model.
IMPORTANTE TUDO QUE CRIARMOS QUE REFERE A CLASSES JAVA DEVEM SER CRIADAS DENTRO DO PACOTE `com.projeto25x.Projetos2x` (onde o x é a final da sua turma)

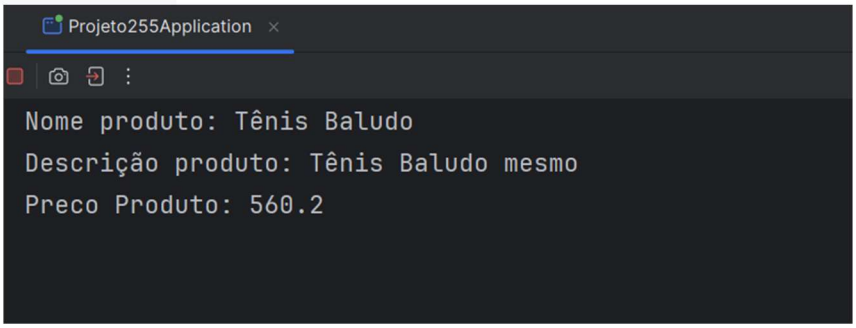
Vamos criar entao Model.Produto

```
public class Produto {
     private Long id;
     private String nome;
     private String descricao;
     private double preco;

     public Long getId() {
          return id;
     }

     public void setId(Long id) {
          this.id = id;
     }

     public String getNome() {
          return nome;
     }

     public void setNome(String nome) {
          this.nome = nome;
     }

     public String getDescricao() {
          return descricao;
     }

     public void setDescricao(String descricao) {
          this.descricao = descricao;
     }

     public double getPreco() {
          return preco;
     }

     public void setPreco(double preco) {
          this.preco = preco;
     }
}
```

Agora vamos criar a classe controle, ou a classe que vai receber as requisições, vamos criar assim Controller.ControleProduto

```
@RestController // Avisa que a classe é responsavel pelas  requisições REST
@RequestMapping("produtos") // Da responsabilidade de  mapeamento das URL para a classe no caso chegue uma chamada "produtos"

public class ProdutoControle {

    @PostMapping

    public void salvar(@RequestBody Produto produto){
        System.out.println("Nome produto: "+ produto.getNome());
        System.out.println("Descrição produto: "+ produto.getDescricao());
        System.out.println("Preco Produto: "+ produto.getPreco());
    }

    /*
    @RequestBody é uma anotação do Spring Framework (no Java) usada principalmente em APIs REST para
    indicar que o corpo da requisição HTTP (normalmente em formato JSON, XML, etc.)
    deve ser convertido automaticamente para um objeto Java.
    */
}
```





JSON é a sigla para JavaScript Object Notation.

É um formato leve e fácil de ler para trocar dados entre sistemas — principalmente entre o frontend (tipo um site) e o backend (como uma API).

Ele é basicamente uma maneira de escrever dados estruturados (como objetos e listas) usando texto.

```
{
    "nome": "Tênis Baludo",
    "descricao": "Tênis Baludo mesmo",
    "preco": 560.20
}
```

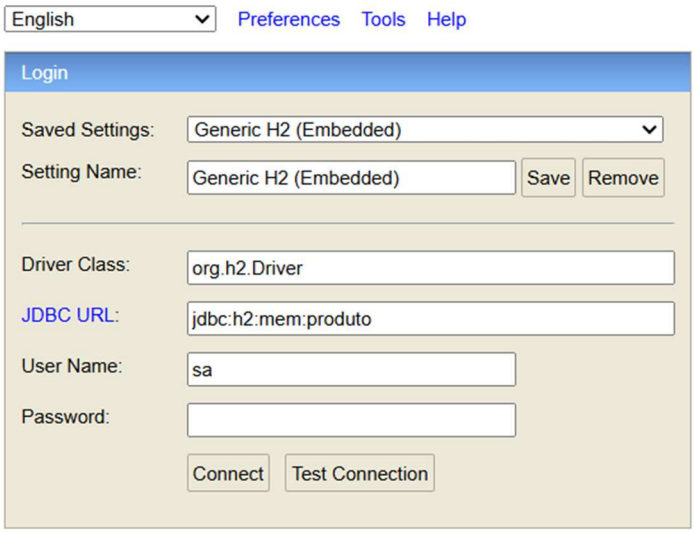
Vamos fazer a conexão com o banco de dados.

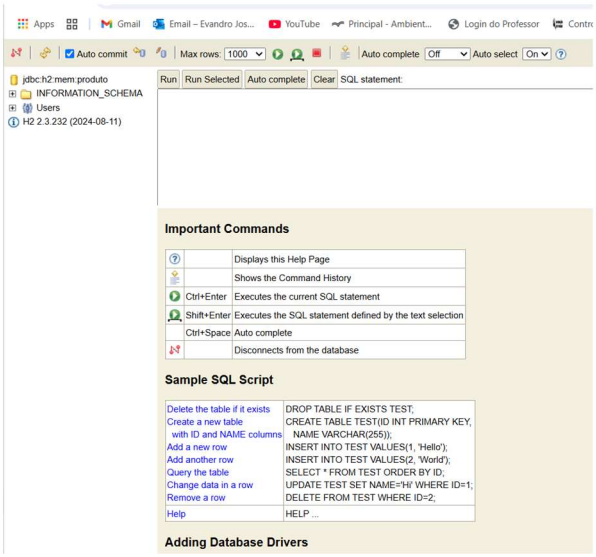
Vamos para a pasta "resource" → Vamos criar o arquivo

"application.yml"

```
spring:
  application:
    name: Projeto255
  datasource:
    url: jdbc:h2:mem:produto
    username: sa
    password:
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
  h2:
    console:
      enabled: true
      path: /banco
```

Vamos testar localhost:8080/banco





Criando a table → dentro da pasta "resources" → Crie data.sql

```
CREATE TABLE produto (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255),
    descricao VARCHAR(255),
    preco decimal(8,2)
);
```

Agora vamos fazer o mapeamento dos dados que serão enviados com a nossa classe Produtos

```java
import jakarta.persistence.*;

@Entity
@Table(name="produto") // Se for mesmo nome não obrigatório
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // ou AUTO, SEQUENCE, TABLE
    @Column(name="id") // Se for mesmo nome não obrigatório
    private Long id;

    @Column(name="nome") // Se for mesmo nome não obrigatório
    private String nome;

    @Column(name="descricao") // Se for mesmo nome não obrigatório
    private String descricao;

    @Column(name="preco") // Se for mesmo nome não obrigatório
    private double preco;

    public Long getId() {
         return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getPreco() {
        return preco;
    }

    public void setPreco(double preco) {
        this.preco = preco;
    }
}
```

Agora vamos acionar o JPA

JPA significa Java Persistence API.

É uma especificação (um conjunto de regras) da linguagem Java para mapear objetos Java para tabelas de banco de dados.

Ou seja:

Você cria classes Java (chamadas de entidades),

E o JPA cuida de salvar, buscar, atualizar e deletar essas classes no banco de dados.

Vamos criar uma Java Classe mas agora vamos usar "Interface"

Dentro do nosso pacote principal vamos fazer a criação

New → Java Class → Interface → Repositorio.ProdutoRepositorio

```java
public interface ProdutoRepositorio extends  JpaRepository<Produto,Long> {

}
```

Vamos alterar a Classe ProdutoControle dentro do pacote Controller

```java
@RestController // Avisa que a classe é responsavel pelas  requisições REST
@RequestMapping("produtos") // Da responsabilidade de  mapeamento das URL para a classe

public class ProdutoControle {

 private ProdutoRepositorio acoesProduto; // Este atributo vamos colocar várias ações
    // que podemos fazer como  produto salvar,consultar.

// Classe constrtutora para iniciar a classe com a acao e mais algum parâmetro

 public ProdutoControle(ProdutoRepositorio produtoRepository) {
        this.acoesProduto = produtoRepository;
    }

// Inclusão de dados vem via Postman
    @PostMapping
    public void salvar(@RequestBody Produto produto){
        System.out.println("Nome produto: "+ produto.getNome());
        System.out.println("Descrição produto: "+ produto.getDescricao());
        System.out.println("Preco Produto: "+ produto.getPreco());
        acoesProduto.save(produto);
    }
    /*
    @RequestBody é uma anotação do Spring Framework (no Java) usada principalmente em APIs REST para
    indicar que o corpo da requisição HTTP (normalmente em formato JSON, XML, etc.)
    deve ser convertido automaticamente para um objeto Java.
    */
```

TRABALHO AVALIATIVO – PARTE 01

Baseado no aprendemos até aqui, vocês deverão criar um banco para cadastrar a marca, modelo e preço de veículos, isso e tudo com vcs. ATENÇÃO NÃO PODERÃO USAR UMA COPIA DO PROGRAMA ACIMA, VCS DEVERÃO CRIAR AS CLASSES, COM ATRIBUTOS E METODOS NOVOS (CRIAÇÃO DE VCS). O que deve ser padrão e o caminho para inclusão qu deve ser "veículos" e o caminho do banco "banco_veiculo". Mande pelo site na Guia TrabAvali com o número da sua turma