



Spring Boot Anotações Spring JPA

- O Spring é um framework Java que tem grande aceitação na área de desenvolvimento de sistemas. Embora não tenha sido lançado como um framework de persistência de dados, ele provê ferramentas que **facilitam** o desenvolvimento das classes que contêm as **operações de CRUD**, tanto com o **uso de JDBC puro**, como com algum **framework de mapeamento objeto relacional (ORM)**.

- O projeto Spring Data JPA, embora não seja um framework ORM, foi desenvolvido com base no padrão JPA 2 para trabalhar com qualquer framework que siga tal especificação.
- Ele é responsável pela implementação dos repositórios (camada de persistência de dados), oferecendo funcionalidades sofisticadas e comuns à maioria dos métodos de acesso a banco de dados.
- Ao programador, se abstrai a necessidade de criar classes concretas para os repositórios, sendo necessário apenas criar uma interface específica para cada classe de entidade, e nelas, estender a interface **JpaRepository**.

- **O Padrão Repository**
- Repository (em português, repositório) é um padrão de projeto descrito no livro Domain-Driven Design (DDD) de Eric Vans.
- É um conceito muito semelhante ao padrão de projeto DAO, já que seu foco também é a camada de persistência de dados de uma aplicação.
- Muitas vezes a implementação de um DAO acaba sendo tratada como um Repository, ou vice-versa, mas estas abordagens se diferem em alguns pontos.

- **O Padrão Repository**
- Os **repositórios são parte da camada de negócios da aplicação**, e fornecem objetos a outras camadas como as de controle ou visão. Outra particularidade do repositório é que ele **não conhece a infraestrutura da aplicação, como o tipo de banco de dados, ou se uma conexão será por JDBC, ODBC** ou mesmo se vai trabalhar com um framework de persistência.
- Já o **padrão DAO conhece a infraestrutura usada e tem a responsabilidade de traduzir as chamadas de persistência em chamadas de infraestrutura**, como preparar os dados que serão persistidos em um banco de dados. O repositório, então, não seria nada mais que uma interface, e poderíamos dizer que as regras existentes nas classes que implementam tais interfaces seriam os DAOs.

- Algumas das principais anotações Spring JPA:
- **@Entity** → define nossa classe como uma entidade de nosso sistema
- **@Table(name = "person")** → define o nome da tabela a ser gerada no banco de dados
- **@Id** → define o atributo identificador da classe.
- **@GeneratedValue(strategy = "...")** → define a estratégia de geração da chave primária da classe.

- Algumas das principais anotações Spring JPA:
- **@GeneratedValue(strategy = "...")** → Por padrão, quando utilizamos somente a anotação **@GeneratedValue** sem passar nenhum argumento, a JPA utiliza a estratégia automática (**GenerationType.AUTO**).
- As estratégias de geração:
- **GenerationType.AUTO** → valor padrão, deixa com o provedor de persistência a escolha da estratégia mais adequada de acordo com o banco de dados.
- **GenerationType.IDENTITY** → informamos ao provedor de persistência que os valores a serem atribuídos ao identificador único serão gerados pela coluna de auto incremento do banco de dados. Assim um valor para o identificador é gerado para cada registro inserido no banco. Alguns bancos de dados podem não suportar essa opção.

- **GenerationType.SEQUENCE** → informamos ao provedor de persistência que os valores gerados a partir de uma sequence. Caso não seja especificado um nome para a sequence, será utilizada uma sequência padrão, a qual será global, para todas as entidades. Caso uma sequence seja especificada, o provedor passará a adotar essa sequência para criação das chaves primárias. Alguns bancos podem não suportar essa opção
- **GenerationType.TABLE** → com a opção Table é necessário criar uma tabela para gerenciar as chaves primárias. Por causa da sobrecarga de consultas necessárias para manter a tabela atualizada, essa opção é pouco recomendada.

- **GenerationType.UUID** → Em relação a essa estratégia o tipo de dado UUID gera uma chave de 128bits serializada para não ocorrer em erro chave repetida em ambiente distribuído. Isto é implementado pela `java.util.UUID` que fornece o tipo de dado UUID que utilizamos em nossa atributo `id`.

```
UUID id;
```

`java.util.UUID`

A class that represents an immutable universally unique identifier (UUID). A UUID represents a 128-bit value.

There exist different variants of these global identifiers. The methods of this class are for manipulating the Leach-Salz variant, although the constructors allow the creation of any variant of UUID (described below).

The layout of a variant 2 (Leach-Salz) UUID is as follows: The most significant long consists of the following unsigned fields:

0xFFFFFFFF00000000	time_low
0x00000000FFFF0000	time_mid
0x000000000000F000	version
0x0000000000000FFF	time_hi

The least significant long consists of the following unsigned fields:

```
1 package com.curso.domains;
2
3 import com.curso.domains.enums.Status;
4 import jakarta.persistence.*;
5 import jakarta.validation.constraints.NotBlank;
6 import jakarta.validation.constraints.NotNull;
7
8 import java.util.Objects;
9
10 @Entity 6 usages  ⚡ jeffersonarpasserini *
11 @Table(name="grupoproduto")
12 public class GrupoProduto {
13
14     @Id 6 usages
15     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_grupoproduto")
16     private int id;
17
18     @NotNull @NotBlank 6 usages
19     private String descricao;
20
21     @Enumerated(EnumType.ORDINAL) 4 usages
22     @JoinColumn(name="status")
23     private Status status;
24
25     public GrupoProduto() { no usages  ⚡ jeffersonarpasserini *
26         this.status = Status.ATIVO;
27     }
```

- **Vamos adicionar as anotações Spring JPA para nossa entidade - GrupoProduto.**
- **@Entity** → define nossa classe como uma entidade de nosso sistema
- **@Table(name = "grupoproduto")** → define o nome da tabela a ser gerada no banco de dados
- **@Id** → define o atributo identificador da classe.
- **@GeneratedValue(strategy = GenerationType.SEQUENCE, generator="seq_grupoproduto")** → define a estratégia de geração da chave da classe GrupoProduto será realizada por uma sequence no banco de dados.

```
1 package com.curso.domains;
2
3 import com.curso.domains.enums.Status;
4 import jakarta.persistence.*;
5 import jakarta.validation.constraints.NotBlank;
6 import jakarta.validation.constraints.NotNull;
7
8 import java.util.Objects;
9
10 @Entity 6 usages  ⬆ jeffersonarpasserini *
11 @Table(name="grupoproduto")
12 public class GrupoProduto {
13
14     @Id 6 usages
15     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_grupoproduto")
16     private int id;
17
18     @NotNull @NotBlank 6 usages
19     private String descricao;
20
21     @Enumerated(EnumType.ORDINAL) 4 usages
22     @JoinColumn(name="status")
23     private Status status;
24
25     public GrupoProduto() { no usages  ⬆ jeffersonarpasserini *
26         this.status = Status.ATIVO;
27     }
```

- **@NotNull** → define que o nosso atributo descrição não pode ser nulo
- **@NotBlank** → define que o nosso atributo descrição não pode ter somente espaços em branco.
- **@Enumerated(EnumType.ORDINAL)** → define que o atributo status será persistido no banco de dados com seu Ordinal, ou seja, se temos valores 0 - Inativo e 1 - Ativo, será gravado 0 ou 1.
- **@JoinColumn(name="status")** → define o nome do campo no banco de dados que irá receber o valor ordinal do Enum.

- Aproveite para definir no construtor que quando gerado um novo grupoproduto ele deve ser gerado como **ATIVO**.

```
24  
25     public GrupoProduto() { no usages  jeffersonarpasserini *  
26         this.status = Status.ATIVO;  
27     }
```

- Resumo das diferenças entre @NotNull, @NotBlank e @NotEmpty

Anotação	Validação	Tipos aplicáveis	Exemplos que falham na validação
@NotNull	Verifica se o valor não é <code>null</code> .	Qualquer tipo (String, List, etc.)	<code>null</code>
@NotEmpty	Verifica se o valor não é <code>null</code> e não está vazio.	Strings, coleções, arrays	<code>null</code> , <code>""</code> , listas vazias <code>[]</code>
@NotBlank	Verifica se o valor não é <code>null</code> , não está vazio e não contém apenas espaços.	Apenas Strings	<code>null</code> , <code>""</code> , <code>" "</code> (apenas espaços)

- Na aula anterior na **classe Produto** definimos nossos atributos **saldoEstoque** e **valorUnitario** como **double**. Vamos alterá-los para o tipo bigdecimal.

```
private BigDecimal saldoEstoque; 3 usages  
private BigDecimal valorUnitario; 3 usages
```

Observação: ficaremos com erros no construtor e nos métodos Get e Set destes atributos. Não se preocupe, faremos a correção adiante no material de aula.

- Aproveite também e já defina o status do produto como Ativo quando gerado um novo objeto de produto, como já fizemos com a classe GrupoProduto.

```
public Produto() { no usages jeffersonar  
    this.status = Status.ATIVO;  
}
```

- **Vamos conhecer as diferenças entre os tipos de dados.**
- A diferença entre **BigDecimal** e **double** em Java é importante, especialmente quando se trata de precisão e uso.
- Cada um tem seus casos de uso específicos, e a escolha entre eles depende do **nível de precisão** necessário e da **natureza dos cálculos**.

- **1. double**
- O **double** é um tipo primitivo em Java que representa um número em ponto flutuante de precisão dupla. Ele usa o padrão IEEE 754 para representar números decimais, o que significa que o valor é armazenado como uma aproximação binária de números reais.
- **Precisão: double tem 64 bits**, com 52 bits para a mantissa (precisão) e o restante para o expoente e o sinal. Isso permite representar um grande intervalo de números, mas como usa ponto flutuante, ele pode perder precisão, especialmente em cálculos com muitos decimais.
- **Velocidade:** Como tipo primitivo, double é rápido e eficiente em termos de desempenho.
- **Limitações:** Ele é suscetível a erros de arredondamento, tornando-o inadequado para cálculos financeiros ou outros que requerem alta precisão. Cálculos como $0.1 + 0.2$ podem resultar em um valor aproximado como 0.30000000000000004.
- **Armazenamento:** Não é adequado para representar valores monetários, pois a imprecisão binária pode causar problemas ao lidar com números decimais.

```
double valor = 0.1 + 0.2;  
System.out.println(valor); // Exibe 0.30000000000000004
```

- **2. BigDecimal**

- **BigDecimal** é uma **classe** da biblioteca Java que oferece uma representação decimal exata para valores numéricos. Ele é projetado para **cálculos que exigem alta precisão e exatidão**, como cálculos financeiros e científicos.
- **Precisão:** BigDecimal oferece precisão arbitrária, ou seja, ele pode representar números com quantos dígitos forem necessários, desde que haja memória suficiente. Isso o torna ideal para situações em que a precisão é crítica.
- **Velocidade:** Como BigDecimal é uma classe de objeto (não um tipo primitivo), ele é mais lento que double em termos de desempenho. Isso ocorre porque BigDecimal realiza operações aritméticas de forma mais complexa e envolve a criação de objetos.
- **Armazenamento:** Armazena números como uma combinação de um número inteiro (mantissa) e uma escala (que define a posição da vírgula). Por isso, ele não sofre com erros de arredondamento como o double.
- **Operações:** BigDecimal **requer o uso de métodos explícitos para realizar operações aritméticas, como .add(), .subtract(), .multiply(), em vez dos operadores aritméticos convencionais +, -, *.**

```
BigDecimal valor1 = new BigDecimal("0.1");
BigDecimal valor2 = new BigDecimal("0.2");
BigDecimal resultado = valor1.add(valor2);
System.out.println(resultado); // Exibe 0.3
```


- Principais diferenças

Característica	<code>double</code>	<code>BigDecimal</code>
Tipo	Primitivo	Objeto (<code>java.math.BigDecimal</code>)
Precisão	Aproximação de ponto flutuante (erro de arredondamento)	Precisão arbitrária, sem erros de arredondamento
Desempenho	Muito rápido	Mais lento (mais processamento e criação de objetos)
Uso	Bom para cálculos científicos e gráficos que não exigem alta precisão	Ideal para cálculos financeiros, monetários ou científicos que exigem precisão
Arredondamento	Sujeito a erros de arredondamento	Controlado e exato, sem erros inesperados
Operações	Operadores aritméticos (<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>)	Métodos (<code>add()</code> , <code>subtract()</code> , <code>multiply()</code> , etc.)
Memória	Ocupa menos memória	Pode ocupar mais memória, especialmente para grandes números ou alta precisão

- **Quando usar cada um**
- **double:** Use double quando o desempenho for uma prioridade e quando não for necessário manter uma precisão exata. Exemplos incluem gráficos, física de jogos e cálculos onde a precisão absoluta não é essencial.
- **BigDecimal:** Use BigDecimal quando a precisão é crítica, como em cálculos financeiros (ex. cálculos de moeda) e aplicações científicas que não podem tolerar erros de arredondamento. Ele é ideal para aplicações onde a exatidão é mais importante que a velocidade.
- **Exemplo de aplicação no mundo financeiro:**
- Quando estamos lidando com valores monetários, como em uma transação de banco ou faturas, BigDecimal é preferido devido à precisão que ele garante, evitando que pequenas discrepâncias se acumulem ao longo do tempo, o que poderia ser catastrófico em grandes somas.

```
1 package com.curso.domains;
2
3 import com.curso.domains.enums.Status;
4 import com.fasterxml.jackson.annotation.JsonFormat;
5 import jakarta.persistence.*;
6 import jakarta.validation.constraints.Digits;
7 import jakarta.validation.constraints.NotBlank;
8 import jakarta.validation.constraints.NotNull;
9
10 import java.math.BigDecimal;
11 import java.time.LocalDate;
12 import java.util.Objects;
13
14 @Entity 2 usages 1 jeffersonarpasserini *
15 @Table(name = "produto")
16 public class Produto {
17
18     @Id 6 usages
19     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_produto")
20     private long idProduto;
21
22     @NotBlank @NotNull 6 usages
23     private String descricao;
24 }
```

- Vamos fazer as anotações da classe **Produto**.
- **@Entity** → define nossa classe como uma entidade de nosso sistema
- **@Table(name = "produto")** → define o nome da tabela a ser gerada no banco de dados

- **@Id** → define o atributo identificador da classe.
- **@GeneratedValue(strategy = GenerationType.SEQUENCE, generator="seq_produto")** → define a estratégia de geração da chave da classe GrupoProduto será realizada por uma sequence no banco de dados.
- **@NotNull** → define que o nosso atributo descrição não pode ser nulo
- **@NotBlank** → define que o nosso atributo descrição não pode ter somente espaços em branco.

- **@Digits(integer=15, fraction=3** → define o número de inteiros e decimais que nosso atributo irá suportar, neste caso 15 inteiros com 3 casas decimais.
- Aproveitamos para criar um novo atributo: **valorEstoque** que representa o valor monetário do estoque do produto.
- Este atributo ele é calculado a partir do saldoEstoque e do valorUnitario assim não necessitamos que informa-lo em uma inclusão por exemplo.

```
25      @NotNull 3 usages
26      @Digits(integer = 15, fraction = 3)
27      private BigDecimal saldoEstoque;
28
29      @NotNull 3 usages
30      @Digits(integer = 15, fraction = 3)
31      private BigDecimal valorUnitario;
32
33      @NotNull no usages
34      @Digits(integer = 15, fraction = 2)
35      private BigDecimal valorEstoque;
```

- **@JsonFormat(pattern="dd/MM/yyyy")** → define o formato de data que será serializado e desserializado no Json para a comunicação com o front-end. A biblioteca `com.fasterxml.jackson.annotation.JsonFormat` é utilizada para habilitar essa anotação.
- **@Enumerated(EnumType.ORDINAL)** → define que o atributo status será persistido no banco de dados com seu Ordinal, ou seja, se temos valores 0 - Inativo e 1 - Ativo, será gravado 0 ou 1.
- **@JoinColumn(name="status")** → define o nome do campo no banco de dados que ira receber o valor ordinal do Enum.

```
32 @JsonFormat(pattern = "dd/MM/yyyy") 3 usages
33 private LocalDate dataCadastro = LocalDate.now();
34
35 @ManyToOne 3 usages
36 @JoinColumn(name="idgrupoproduto")
37 private GrupoProduto grupoProduto;
38
39 @Enumerated(EnumType.ORDINAL) 4 usages
40 @JoinColumn(name="status")
41 private Status status;
```

- **@ManyToOne** → O relacionamento **@ManyToOne** com a anotação **@JoinColumn** em JPA define uma associação de **muitos para um entre duas entidades**. No caso do código a entidade Produto está relacionada com a entidade GrupoProduto dessa maneira;
- A anotação **@ManyToOne** indica que muitos objetos da entidade **Produto** estão associados a um objeto da entidade **GrupoProduto**.
- **@JoinColumn(name="idgrupoproduto")** → A anotação **@JoinColumn** define a coluna na tabela produto que será usada como **chave estrangeira** para se referir à tabela grupoproduto.
- O **parâmetro name="idgrupoproduto"** indica que a **coluna no banco de dados que armazenará essa chave estrangeira se chama idgrupoproduto**.

```
32 @JsonFormat(pattern = "dd/MM/yyyy") 3 usages
33 private LocalDate dataCadastro = LocalDate.now();
34
35 @ManyToOne 3 usages
36 @JoinColumn(name="idgrupoproduto")
37 private GrupoProduto grupoProduto;
38
39 @Enumerated(EnumType.ORDINAL) 4 usages
40 @JoinColumn(name="status")
41 private Status status;
```

```
public Produto() {
    this.saldoEstoque = BigDecimal.ZERO;
    this.valorUnitario = BigDecimal.ZERO;
    this.valorEstoque = BigDecimal.ZERO;
    this.status = Status.ATIVO;
}

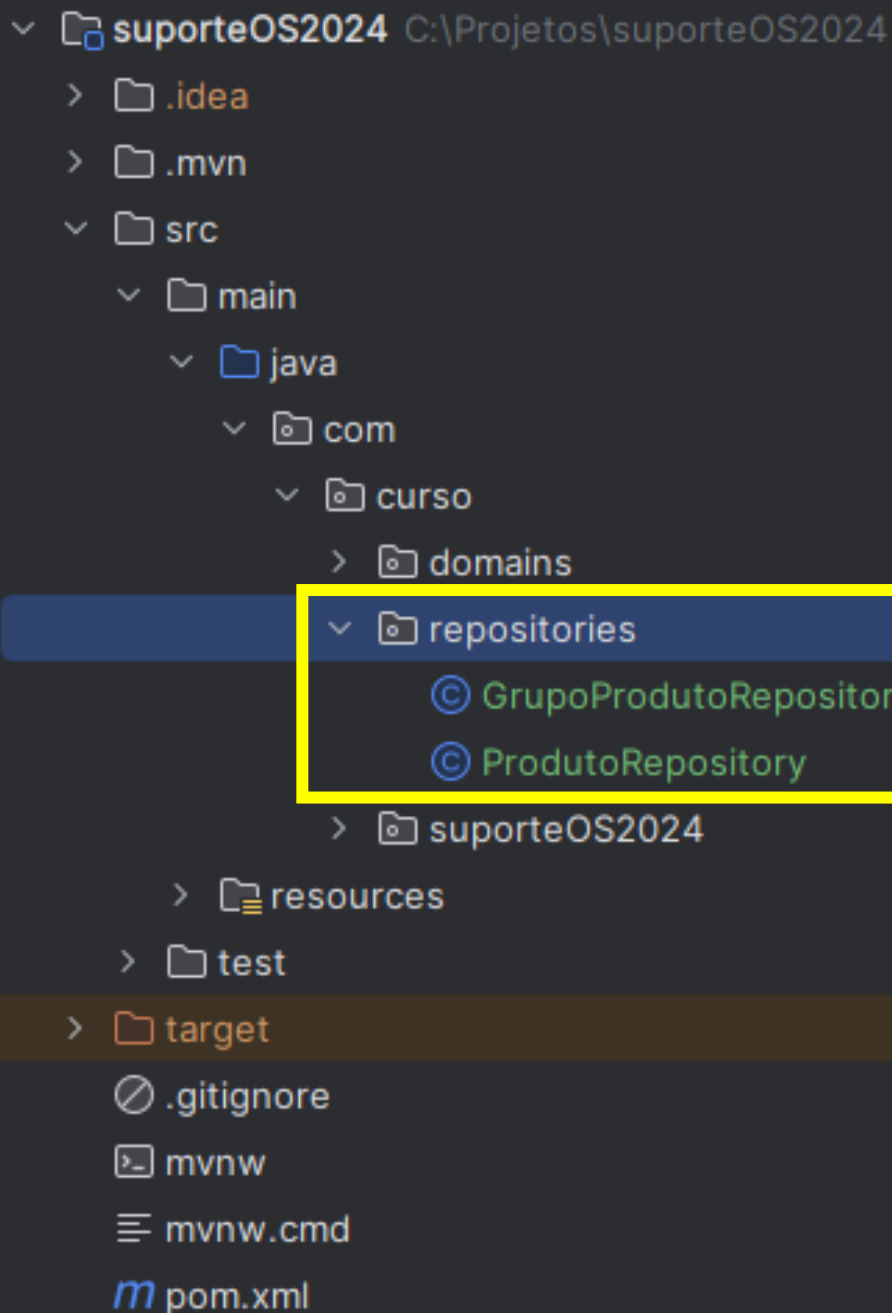
public Produto(long idProduto, String descricao, BigDecimal saldoEstoque, BigDecimal valorUnitario,
               LocalDate dataCadastro, GrupoProduto grupoProduto, Status status) {
    this.idProduto = idProduto;
    this.descricao = descricao;
    //this.saldoEstoque = saldoEstoque;
    this.valorUnitario = valorUnitario;
    this.dataCadastro = dataCadastro;
    this.grupoProduto = grupoProduto;
    this.status = status;

    this.saldoEstoque = saldoEstoque != null ? saldoEstoque : BigDecimal.ZERO;
    this.valorEstoque = saldoEstoque != null ? saldoEstoque.multiply(valorUnitario) : BigDecimal.ZERO;
}
```

- Gere novamente o construtor com parâmetros, mas lembre-se que **não precisamos do atributo valorEstoque no construtor** pois o mesmo é calculado.
- **Mas devemos realizar o cálculo deste atributo, em ambos os construtores.**
- **Gere novamente os métodos Getter and Setter de todos os atributos.**



Spring Boot Spring JPA – Criando os Repositories



- A partir do pacote raiz de nossa aplicação "**curso**" vamos criar o pacote **repositories**.

- Vamos criar os Repository, os **repositores são do tipo interface e estendem o JpaRepository<>**

GrupoProdutoRepository

ProdutoRepository

```
1 package com.curso.repositories;  
2  
3 import com.curso.domains.GrupoProduto;  
4 import org.springframework.data.jpa.repository.JpaRepository;  
5 import org.springframework.stereotype.Repository;  
6  
7 @Repository no usages new *  
8 public interface GrupoProdutoRepository extends JpaRepository<GrupoProduto, Integer> {}  
9  
10 }
```

- Assinalamos nossa interface como @Repository para que o Spring possa trata-la adequadamente.
- Como pode-se observar o JpaRepository<> recebe dois parâmetros que são a classe que o repository irá gerenciar e o tipo de dado da chave da classe, neste caso: JpaRepository<GrupoProduto, Integer>
- **Se você observar no atributo Id da class GrupoProduto é do tipo primitivo int, aqui colocamos o tipo da chave como Integer pois o Spring Data preferem a versão wrapper (Integer ao invés de int) porque permite trabalhar com tipos que podem ser nulos e oferece mais flexibilidade nas operações.**

```
1 package com.curso.repositories;  
2  
3 import com.curso.domains.Produto;  
4 import org.springframework.data.jpa.repository.JpaRepository;  
5 import org.springframework.stereotype.Repository;  
6  
7 @Repository no usages new *  
8 public interface ProdutoRepository extends JpaRepository<Produto, Long> {}  
9  
10 }
```

- Assinalamos nossa interface como @Repository para que o Spring possa trata-la adequadamente.
- Como pode-se observar o JpaRepository<> recebe dois parâmetros que são a classe que o repository irá gerenciar e o tipo de dado da chave da classe, neste caso: JpaRepository<Produto, Long>
- **Se você observar no atributo Id da class Produto é do tipo primitivo long, aqui colocamos o tipo da chave como Long pois o Spring Data preferem a versão wrapper (Long ao invés de long) porque permite trabalhar com tipos que podem ser nulos e oferece mais flexibilidade nas operações.**

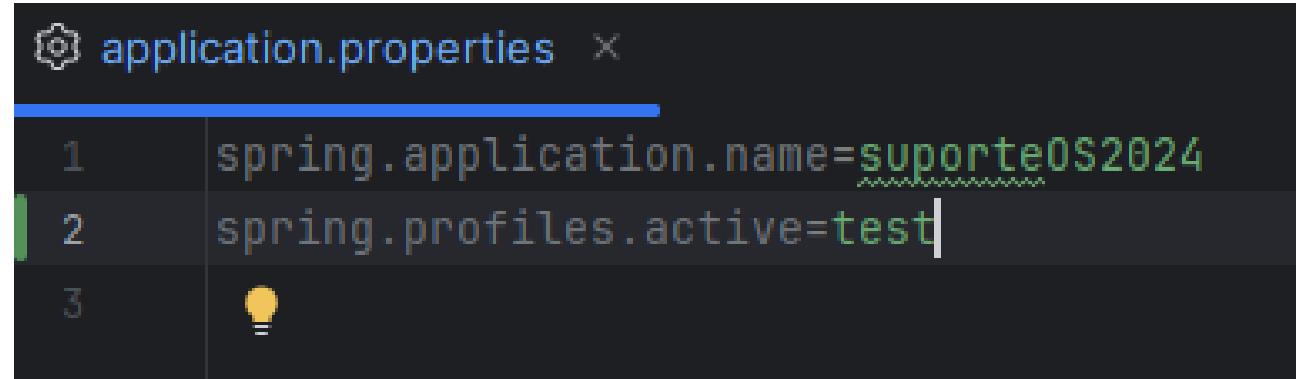


Spring Boot

Spring JPA – Conexão com o BD

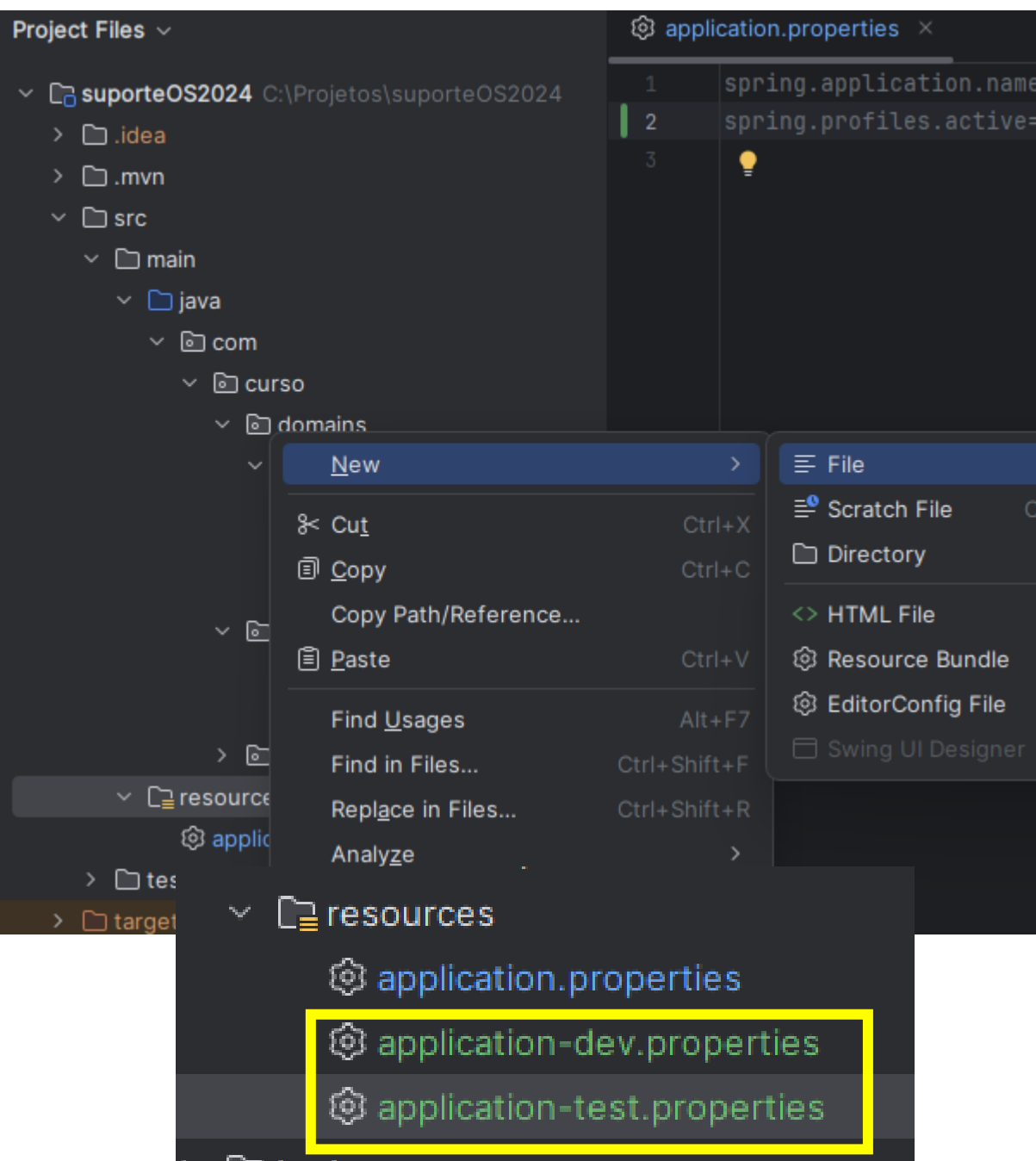
Criando Projeto Spring Boot – Conexão BD

- Vamos preparar as configurações para que nosso projeto tenha acesso ao banco de dados.
- Para isso iremos utilizar o arquivo **application.properties** que se localiza em **resources**.
- Serão criados perfis para que nossa aplicação possa rodar com diferentes configurações como em um ambiente de teste ou de desenvolvimento.
- Cada um das configurações serão representadas por dois arquivos de configuração independentes onde iremos atribuir as configurações dos banco de dados - PostgreSQL e H2 Database.
- Com isso temos dois perfis de execução para nosso sistema o que é interessante quando temos que testar a aplicação em diferentes situações.



```
application.properties x
1 spring.application.name=suporteOS2024
2 spring.profiles.active=test
3
```

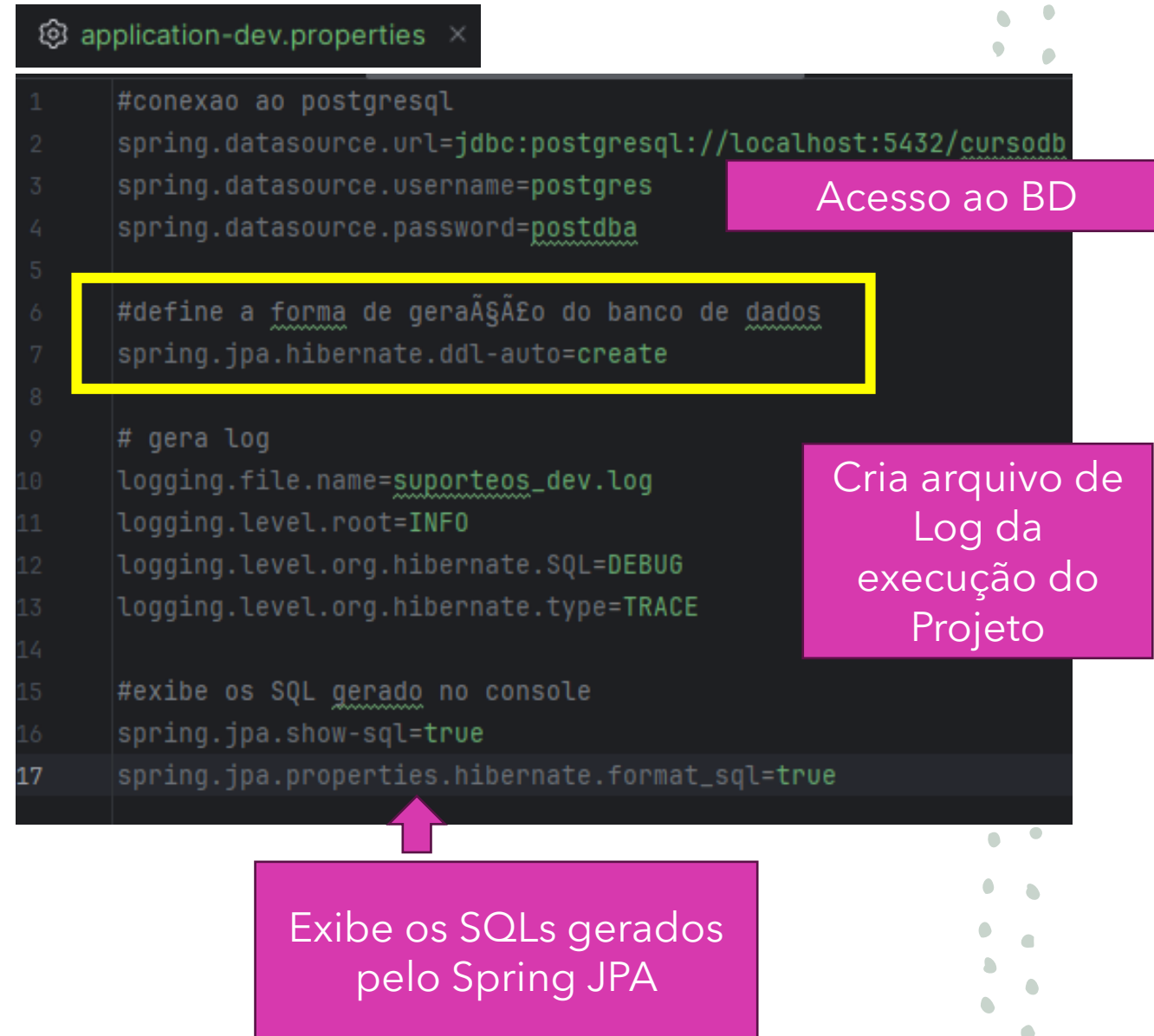
- Coloque em nosso **application.properties** a linha acima.
- → **spring.profiles.active=test**
- Assim estamos definindo para o Spring qual **Profile** de projeto desejamos executar.



Criando Projeto Spring Boot – Conexão BD

- Clique com o botão direito em resources e escolha Novo Arquivo.
- Adicione os arquivos:
- **application-dev.properties**
- **application-test.properties**

- Edite o arquivo **application-dev.properties**
- Insira as configurações que serão carregadas quando o **Profile** do projeto estiver definido como **dev**.
- **Inicialmente (linhas 2-4) temos a definição da string de conexão com o nome do banco de dados a ser utilizado.**
- **Assim como temos o usuário e a senha para acesso.**



The image shows a code editor window titled 'application-dev.properties'. The file contains the following configuration lines:

```
1 #conexao ao postgresql
2 spring.datasource.url=jdbc:postgresql://localhost:5432/cursodb
3 spring.datasource.username=postgres
4 spring.datasource.password=postgresdb
5
6 #define a forma de geração do banco de dados
7 spring.jpa.hibernate.ddl-auto=create
8
9 # gera log
10 logging.file.name=suporteos_dev.log
11 logging.level.root=INFO
12 logging.level.org.hibernate.SQL=DEBUG
13 logging.level.org.hibernate.type=TRACE
14
15 #exibe os SQL gerado no console
16 spring.jpa.show-sql=true
17 spring.jpa.properties.hibernate.format_sql=true
```

Annotations in pink boxes:

- Acesso ao BD**: Points to lines 2-4, which define the database connection URL, username, and password.
- Cria arquivo de Log da execução do Projeto**: Points to lines 9-13, which configure logging to a file and set log levels for the root and Hibernate SQL.
- Exibe os SQLs gerados pelo Spring JPA**: Points to lines 15-17, which enable SQL display in the console and format the SQL output.

- A configuração **spring.jpa.hibernate.ddl-auto=create** pode assumir:
- **create** --> cria o esquema destruindo os anteriores
- **create-drop** --> cria o esquema ao iniciar e descarta ao fechar a aplicação
- **update** --> atualiza o esquema mantendo os dados existentes e fazendo as alterações necessárias
- **validate** --> valida o esquema, fazendo com que a aplicação falhe ao iniciar se o esquema não estiver correto
- **none** → não faz nada

```
application-dev.properties x
1  #conexao ao postgresql
2  spring.datasource.url=jdbc:postgresql://localhost:5432/cursodb
3  spring.datasource.username=postgres
4  spring.datasource.password=postdba
5
6  #define a forma de geração do banco de dados
7  spring.jpa.hibernate.ddl-auto=create
8
9  # gera log
10 logging.file.name=suporteos_dev.log
11 logging.level.root=INFO
12 logging.level.org.hibernate.SQL=DEBUG
13 logging.level.org.hibernate.type=TRACE
14
15 #exibe os SQL gerado no console
16 spring.jpa.show-sql=true
17 spring.jpa.properties.hibernate.format_sql=true
```

Acesso ao BD

Cria arquivo de Log da execução do Projeto

Exibe os SQLs gerados pelo Spring JPA

- Nas linhas 12 a 16 temos a definição de um arquivo com o log da execução para que possamos depurar erros de nossa aplicação
- Nas linha 18 a 20 informamos o jpa para exibir os códigos SQL gerados.
- Lembre-se de criar o banco de dados **kursodb** no PostgreSQL antes de rodar seu projeto em modo **dev**.



The image shows a code editor window titled 'application-dev.properties'. The code is as follows:

```
1 #conexao ao postgresql
2 spring.datasource.url=jdbc:postgresql://localhost:5432/kursodb
3 spring.datasource.username=postgres
4 spring.datasource.password=postgres
5
6 #define a forma de geração do banco de dados
7 spring.jpa.hibernate.ddl-auto=create
8
9 # gera log
10 logging.file.name=suporteos_dev.log
11 logging.level.root=INFO
12 logging.level.org.hibernate.SQL=DEBUG
13 logging.level.org.hibernate.type=TRACE
14
15 #exibe os SQL gerado no console
16 spring.jpa.show-sql=true
17 spring.jpa.properties.hibernate.format_sql=true
```

Annotations (callouts) are present:

- A pink box labeled 'Acesso ao BD' points to lines 2-4.
- A pink box labeled 'Cria arquivo de Log da execução do Projeto' points to lines 9-13.
- A pink box labeled 'Exibe os SQLs gerados pelo Spring JPA' points to lines 16-17.

- Edite o arquivo **application-test.properties**
- Insira as configurações que serão carregadas quando o **Profile** do projeto estiver definido como **test (H2)**.
- O banco de dados H2 roda em memória durante o período que o projeto está sendo executado.
- **spring.h2.console.enabled=true** → habilita o H2 em seu projeto.
- **spring.h2.console.path=/h2-console** → define o path de acesso ao banco de dados.

```
application-test.properties x
1 #habilitando o banco de dados h2
2 spring.h2.console.enabled=true
3 #path para acesso
4 spring.h2.console.path=/h2-console
5
6 #configuracao acesso ao banco --> jdbc:h2:file:~/cursodb
7 spring.datasource.url=jdbc:h2:mem:cursodb
8 spring.datasource.username=sa
9 spring.datasource.password=
10 spring.datasource.driver-class-name=org.h2.Driver
11
12 # application.properties
13 logging.file.name=suporteos_test.log
14 logging.level.root=INFO
15 logging.level.org.hibernate.SQL=DEBUG
16 logging.level.org.hibernate.type=TRACE
17
18 #exibe os SQL gerado no console
19 spring.jpa.show-sql=true
20 spring.jpa.properties.hibernate.format_sql=true
```

Acesso ao BD

Cria arquivo de Log da execução do Projeto

Exibe os SQLs gerados pelo Spring JPA

- **spring.datasource.url=jdbc:h2:mem:curso**
db → define que o banco cursodb será criado na memória. Se tivermos problemas com essa abordagem podemos optar pela configuração: **jdbc:h2:file:~/cursodb**
- **spring.datasource.username=sa** → nome do usuário
- **spring.datasource.password=** → senha
- **spring.datasource.driver-class-name** → org.h2.Driver

```
application-test.properties x
1 #habilitando o banco de dados h2
2 spring.h2.console.enabled=true
3 #path para acesso
4 spring.h2.console.path=/h2-console
5
6 #configuracao acesso ao banco --> jdbc:h2:file:~/cursodb
7 spring.datasource.url=jdbc:h2:mem:cursodb
8 spring.datasource.username=sa
9 spring.datasource.password=
10 spring.datasource.driver-class-name=org.h2.Driver
11
12 # application.properties
13 logging.file.name=suporteos_test.log
14 logging.level.root=INFO
15 logging.level.org.hibernate.SQL=DEBUG
16 logging.level.org.hibernate.type=TRACE
17
18 #exibe os SQL gerado no console
19 spring.jpa.show-sql=true
20 spring.jpa.properties.hibernate.format_sql=true
```

Acesso ao BD

Cria arquivo de Log da execução do Projeto

Exibe os SQLs gerados pelo Spring JPA

- Nas linhas 12 a 16 temos a definição de um arquivo com o log da execução para que possamos depurar erros de nossa aplicação
- Nas linha 18 a 20 informamos o jpa para exibir os códigos SQL gerados.

```
application-test.properties x
1 #habilitando o banco de dados h2
2 spring.h2.console.enabled=true
3 #path para acesso
4 spring.h2.console.path=/h2-console
5
6 #configuracao acesso ao banco --> jdbc:h2:file:~/cursodb
7 spring.datasource.url=jdbc:h2:mem:cursodb
8 spring.datasource.username=sa
9 spring.datasource.password=
10 spring.datasource.driver-class-name=org.h2.Driver
11
12 # application.properties
13 logging.file.name=suporteos_test.log
14 logging.level.root=INFO
15 logging.level.org.hibernate.SQL=DEBUG
16 logging.level.org.hibernate.type=TRACE
17
18 #exibe os SQL gerado no console
19 spring.jpa.show-sql=true
20 spring.jpa.properties.hibernate.format_sql=true
```

Acesso ao BD

Cria arquivo de Log da execução do Projeto

Exibe os SQLs gerados pelo Spring JPA

Criando Projeto Spring Boot – Conexão BD

```
1 package com.curso.suporteOS2024;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.boot.autoconfigure.domain.EntityScan;
6
7 @EntityScan(basePackages = {"com.curso.domains", "com.curso.domains.enums"})
8 @SpringBootApplication
9 public class SuporteOs2024Application {
10
11     public static void main(String[] args) { SpringApplication.run(SuporteOs2024Application.class, args); }
12
13
14
15 }
```

- Adicione a annotation **@EntityScan** em nossa classe principal como observado na figura.
- Essa anotação fará com que o Spring identifique nossas entidades para o Spring JPA.

```
        descricao varchar(255) not null,  
        primary key (id_produto)  
    )  
Hibernate:  
    create table produto (  
        data_cadastro date,  
        idgrupoproduto integer,  
        saldo_estoque numeric(18,3) not null,  
        status tinyint check (status between 0 and 1),  
        valor_estoque numeric(17,2) not null,  
        valor_unitario numeric(18,3) not null,  
        id_produto bigint not null,  
        descricao varchar(255) not null,  
        primary key (id_produto)  
    )  
2024-09-04T11:25:16.237-03:00 DEBUG 14264 --- [suporte0S2024] [  
    alter table if exists produto  
        add constraint FKdupgm9x214hsg0i5cpu30dj16  
        foreign key (idgrupoproduto)  
        references grupoproduto  
Hibernate:  
    alter table if exists produto  
        add constraint FKdupgm9x214hsg0i5cpu30dj16  
        foreign key (idgrupoproduto)  
        references grupoproduto  
2024-09-04T11:25:16.241-03:00 TRACE 14264 --- [suporte0S2024] [  
2024-09-04T11:25:16.242-03:00 INFO 14264 --- [suporte0S2024] [  

```

- Para acessar o banco digite no navegador:

- **`http://localhost:8080/h2-console`**

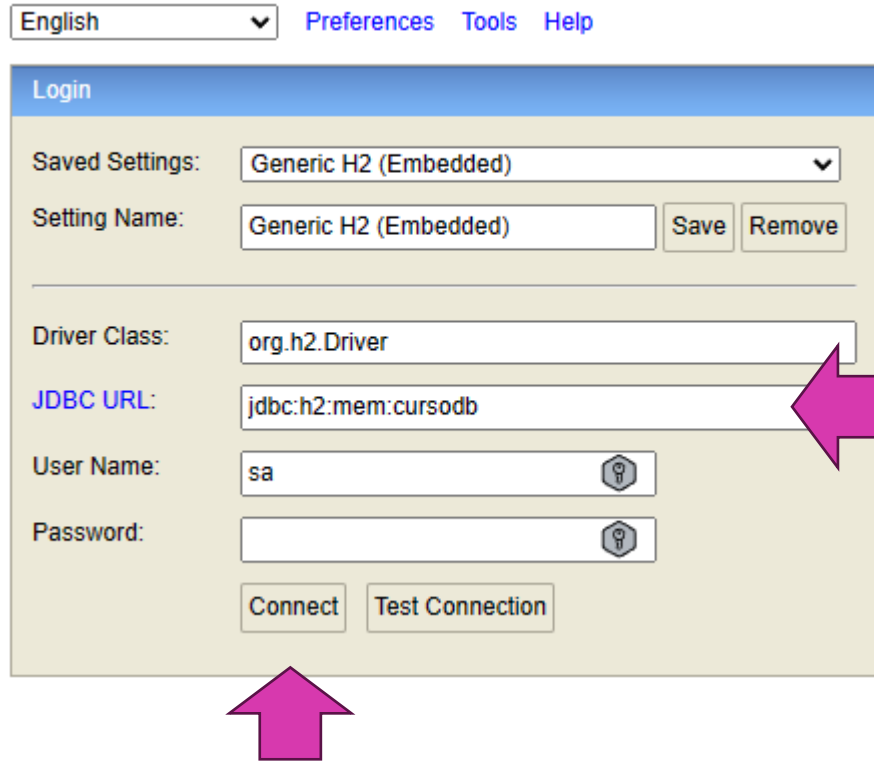
- ***O acesso foi definido no arquivo `application-test.properties`***



```
src > main > resources > application-test.properties  
1  #habilitando o banco de dados h2  
2  spring.h2.console.enabled=true  
3  #path para acesso  
4  spring.h2.console.path=/h2-console
```

- Edite o arquivo **`application.properties`**
- Defina a inicialização como **test e execute seu projeto**
- O Banco de Dados H2 irá iniciar junto com o projeto.

- Tela de acesso ao H2.



- Atenção ao JDBC URL:
- Deve ser como foi definido no arquivo `application-test.properties`***

```
spring.datasource.url=jdbc:h2:mem:cursodb
```

- Conecte-se ao banco de dados, o usuário e a senha também foram definidos nas propriedades.

- Interface do H2 Database

The screenshot shows the H2 Database GUI interface. The left sidebar contains a tree view of the database schema, including tables like GRUPOPRODUTO, PRODUTO, and sequences like SEQ_GRUPOPRODUTO and SEQ_PRODUTO. The main area displays 'Important Commands' and 'Sample SQL Script'.

Important Commands

Icon	Command	Description
?		Displays this Help Page
📜		Shows the Command History
▶	Ctrl+Enter	Executes the current SQL statement
▶	Shift+Enter	Executes the SQL statement defined by the text selection
▶	Ctrl+Space	Auto complete
🔌		Disconnects from the database

Sample SQL Script

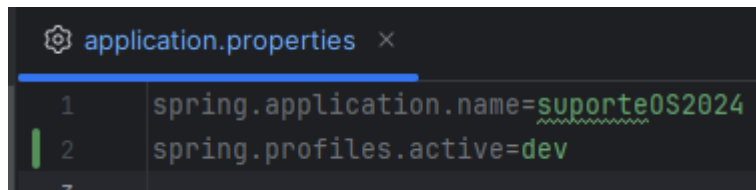
Action	SQL Statement
Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;
Change data in a row	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Remove a row	DELETE FROM TEST WHERE ID=2;
Help	HELP ...

Adding Database Drivers

Additional database drivers can be registered by adding the Jar file location of the driver to the environment variables H2DRIVERS or CLASSPATH. Example (Windows): to add the database driver library C:/Programs/hsaldb/lib/hsaldb.jar set the environment variable H2DRIVERS to C:/Programs/hsaldb/lib/hsaldb.jar

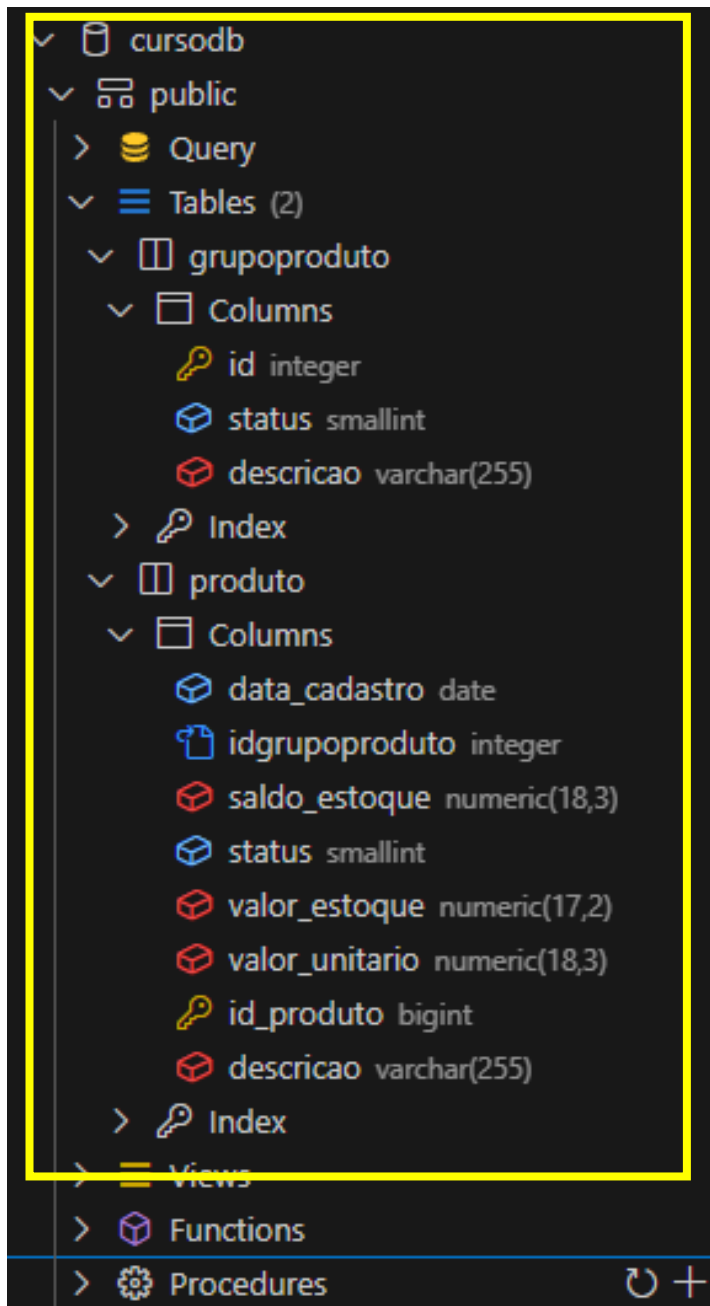
- No destaque pode-se observar as tabelas geradas assim como as sequencias**

- Agora edite novamente o arquivo **application-test.properties**
- Insira as configurações que serão carregadas quando o **Profile** do projeto estiver definido como **dev (PostgreSQL)**.

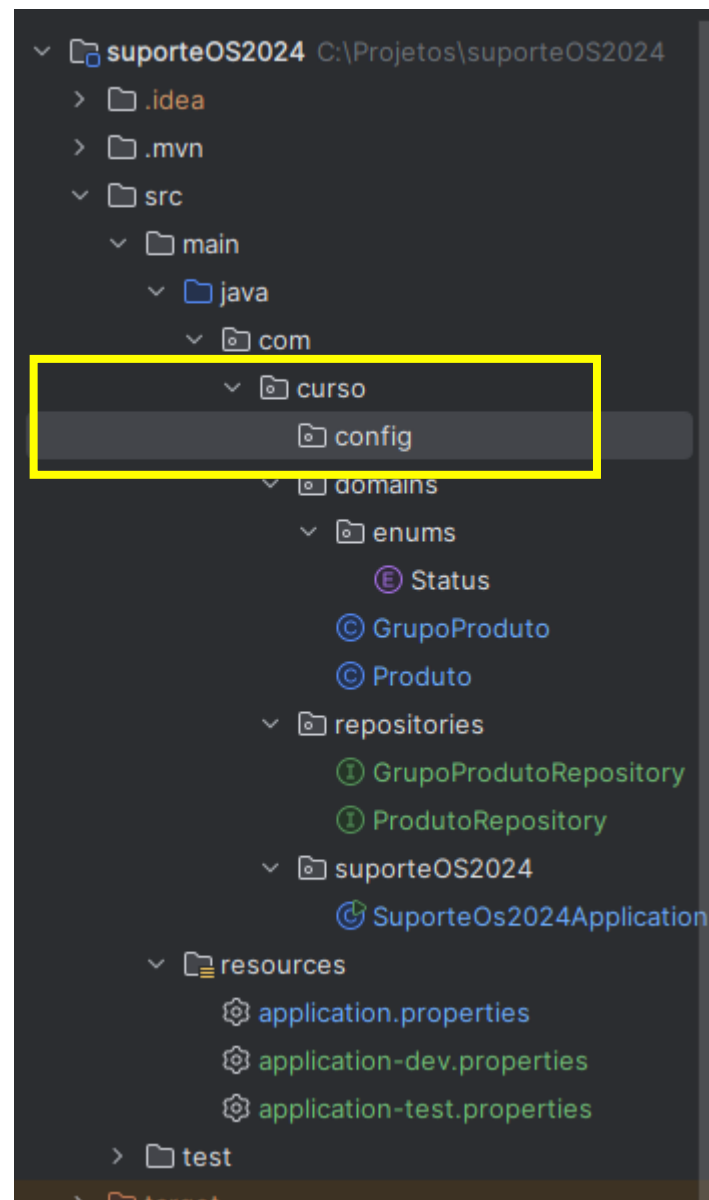


```
1 spring.application.name=suporte0S2024
2 spring.profiles.active=dev
```

- O projeto irá executar e gerar as tabelas no banco de dados **corsodb** no postgresql.
- Não se esqueça que corsodb já deve estar criado no PostgreSQL

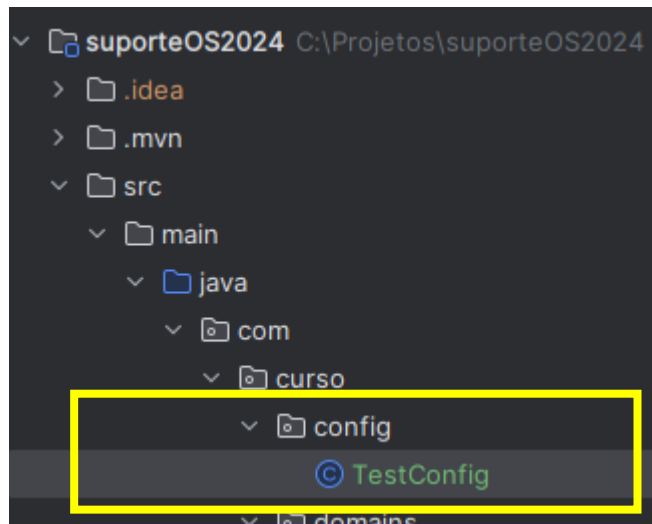


- Para exemplificar o poder dos Profiles vamos criar uma classe que irá gerar uma carga inicial de dados quando a execução estiver configurada como **test**.
- Para o banco de dados H2.
- Para isso vamos criar um novo pacote denominado **config**, a partir do pacote principal de nosso projeto (curso)



Criando Projeto Spring Boot – Conexão BD

- No pacote **config** crie a classe java denominada **TestConfig**.



```
TestConfig.java x
1 package com.curso.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.context.annotation.Profile;
5
6 @Configuration no usages new *
7 @Profile("test")
8 public class TestConfig {
9
10
11 }
12
```

- Faça as anotações **@Configuration** e **@Profile**



-

```
@Service no usages new *
```

-

- Desenvolva o código em **DBService**.

Criando Projeto Spring Boot – Conexão BD

- **@Autowired**
- Passa o controle da injeção de dependência para o Spring.
- Assim não precisamos gerar manualmente uma instância do objeto das classes **Repository**.
- O Spring cuida da tarefa.

```
1 package com.curso.services;
2
3 import com.curso.domains.GrupoProduto;
4 import com.curso.domains.Produto;
5 import com.curso.repositories.GrupoProdutoRepository;
6 import com.curso.repositories.ProdutoRepository;
7 import com.curso.domains.enums.Status;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.stereotype.Service;
10
11 import java.math.BigDecimal;
12 import java.time.LocalDate;
13
14 @Service
15 public class DBService {
16
17     @Autowired
18     private GrupoProdutoRepository grupoProdutoRepo;
19
20     @Autowired
21     private ProdutoRepository produtoRepo;
22
23 }
```

- Desenvolva o código em **DBService**.

```
14  @Service no usages new *
15  public class DBService {
16
17      @Autowired 2 usages
18      private GrupoProdutoRepository grupoProdutoRepo;
19
20      @Autowired 4 usages
21      private ProdutoRepository produtoRepo;
22
23      public void initDB(){ no usages new *
24
25          GrupoProduto grupo01 = new GrupoProduto( id: 0, descricao: "Limpeza", Status.ATIVO);
26          GrupoProduto grupo02 = new GrupoProduto( id: 0, descricao: "Alimenticio", Status.ATIVO);
27
28          Produto produto01 = new Produto( idProduto: 0, descricao: "Coca Cola", new BigDecimal( val: "100"), new BigDecimal( val: "3.5"),
29              LocalDate.now(), grupo02, Status.ATIVO);
30          Produto produto02 = new Produto( idProduto: 0, descricao: "Guarana Antartica", new BigDecimal( val: "200"), new BigDecimal( val: "3.0"),
31              LocalDate.now(), grupo02, Status.ATIVO);
32          Produto produto03 = new Produto( idProduto: 0, descricao: "Detergente Limpol", new BigDecimal( val: "300"), new BigDecimal( val: "4.0"),
33              LocalDate.now(), grupo01, Status.ATIVO);
34          Produto produto04 = new Produto( idProduto: 0, descricao: "Sabão em Pó OMO", new BigDecimal( val: "400"), new BigDecimal( val: "15.5"),
35              LocalDate.now(), grupo02, Status.ATIVO);
36
37          grupoProdutoRepo.save(grupo01);
38          grupoProdutoRepo.save(grupo02);
39          produtoRepo.save(produto01);
40          produtoRepo.save(produto02);
41          produtoRepo.save(produto03);
42          produtoRepo.save(produto04);
43      }
44  }
45  }
```

Você não escreveu nenhuma linha de código nos Repositories mas ele já é capaz de salvar os dados no BD

- Agora desenvolva o código de **TestConfig**.

```
1 package com.curso.config;
2
3 import com.curso.services.DBService;
4 import jakarta.annotation.PostConstruct;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.context.annotation.Profile;
8
9 @Configuration no usages new *
10 @Profile("test")
11 public class TestConfig {
12
13     @Autowired 1 usage
14     private DBService dbService;
15
16     @PostConstruct no usages new *
17     public void initDB(){
18         this.dbService.initDB();
19     }
20 }
```

Criando Projeto Spring Boot – Conexão BD

- Definimos a classe **TestConfig** como:
- **@Configuration** → define para o Spring que essa classe é de configuração.
- **@Profile("test")** → define quando essa classe de configuração será acionada, neste caso apenas quando a propriedade estiver como **test**.

```
≡ application.properties ●
src > main > resources > ≡ application.properties
You, há 5 horas | 1 author (You)
1 spring.profiles.active=test
2
3
```

- Agora desenvolva o código de **TestConfig**.

```
1 package com.curso.config;
2
3 import com.curso.services.DBService;
4 import jakarta.annotation.PostConstruct;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.context.annotation.Profile;
8
9 @Configuration no usages new *
10 @Profile("test")
11 public class TestConfig {
12
13     @Autowired 1 usage
14     private DBService dbService;
15
16     @PostConstruct no usages new *
17     public void initDB(){
18         this.dbService.initDB();
19     }
20 }
```

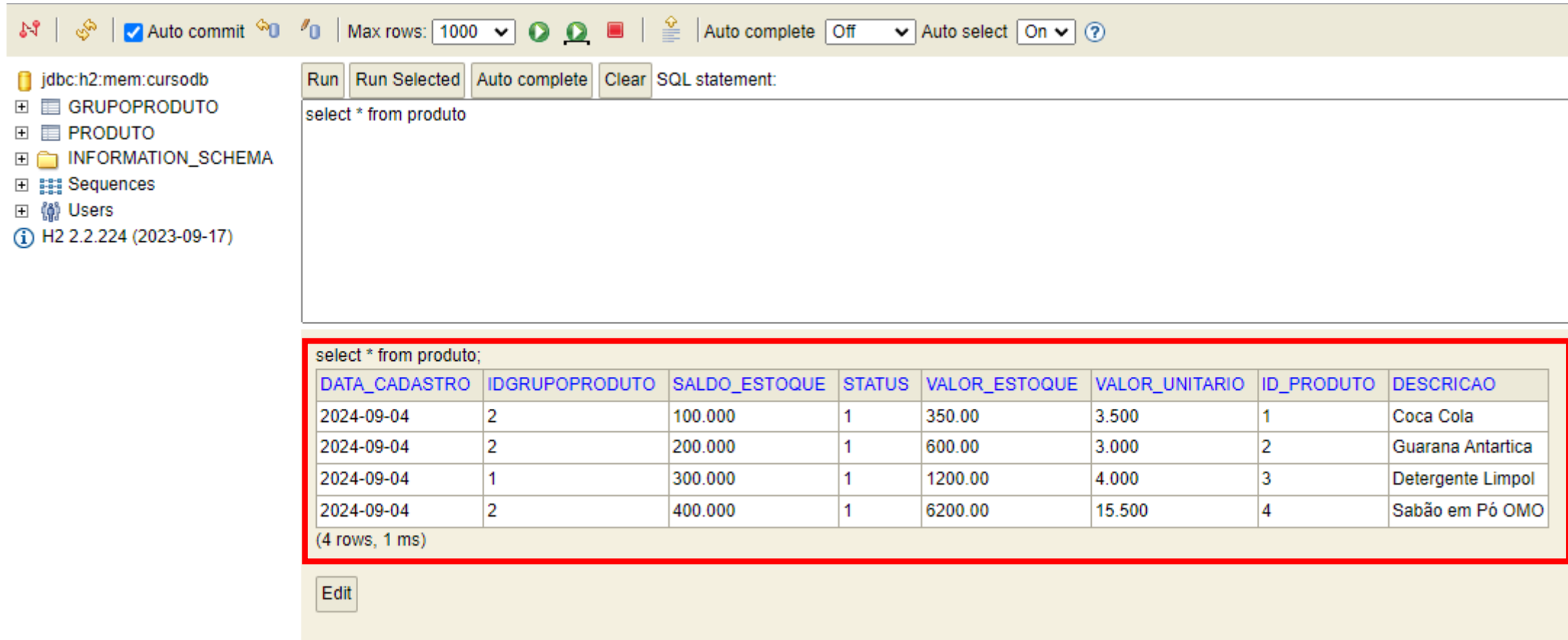
- **@PostConstruct**: Esta anotação faz parte das especificações Java (JSR-250) e **indica que o método anotado deve ser executado logo após a construção do bean e a injeção de todas as suas dependências**.
- Quando o Spring cria o TestConfig e injeta o DBService, o Spring automaticamente chama o método initDB() logo após a inicialização do bean.

```
application.properties
src > main > resources > application.properties
You, há 5 horas | 1 author (You)
1 spring.profiles.active=test
2
3
```


Criando Projeto Spring Boot – Conexão BD

```
1 package com.curso.suporte0S2024;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.boot.autoconfigure.domain.EntityScan;
6 import org.springframework.context.annotation.ComponentScan;
7 import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
8
9 @ComponentScan(basePackages = "com.curso")
10 @EntityScan(basePackages = {"com.curso.domains", "com.curso.domains.enums"})
11 @EnableJpaRepositories(basePackages = "com.curso.repositories")
12 @SpringBootApplication
13 public class Suporte0s2024Application {
14
15     public static void main(String[] args) { SpringApplication.run(Suporte0s2024Application.class, args); }
16
17 }
18
19 }
```

- Confira as annotations da classe principal do software **Suporte0sApplication**.
- **@EntityScan** → Define os pacotes onde estão nossos domínios de dados.
- **@ComponentScan** → define a o pacote inicial para a busca dos componentes de nosso software.
- **@EnableJpaRepositories** → define o pacote dos repositories



The screenshot displays the H2 Database Console interface. The top toolbar includes icons for connection, auto-commit, max rows (set to 1000), and execution controls. The left sidebar shows the database structure: jdbc:h2:mem: cursodb, with expanded nodes for GRUPOPRODUTO, PRODUTO, INFORMATION_SCHEMA, Sequences, and Users. The main SQL statement area contains the query: `select * from produto`. Below the query, the results are displayed in a table format, enclosed in a red border. The table has 8 columns: DATA_CADASTRO, IDGRUPOPRODUTO, SALDO_ESTOQUE, STATUS, VALOR_ESTOQUE, VALOR_UNITARIO, ID_PRODUTO, and DESCRICAO. It contains 4 rows of data. Below the table, it indicates "(4 rows, 1 ms)". An "Edit" button is located at the bottom left of the results area.

Run Run Selected Auto complete Clear SQL statement:

```
select * from produto
```

DATA_CADASTRO	IDGRUPOPRODUTO	SALDO_ESTOQUE	STATUS	VALOR_ESTOQUE	VALOR_UNITARIO	ID_PRODUTO	DESCRICAO
2024-09-04	2	100.000	1	350.00	3.500	1	Coca Cola
2024-09-04	2	200.000	1	600.00	3.000	2	Guarana Antartica
2024-09-04	1	300.000	1	1200.00	4.000	3	Detergente Limpol
2024-09-04	2	400.000	1	6200.00	15.500	4	Sabão em Pó OMO

(4 rows, 1 ms)

Edit

- **Informações inseridas com sucesso**

- **Vamos fazer a carga inicial para o PostgreSQL**
- Como vimos anteriormente o parâmetro **"create"** cria novamente o banco de dados quando iniciamos nosso projeto.
- Mas se já houver dados no banco não seria o correto utilizarmos dessa forma.
- Altere o **spring.profiles.active = dev** em application.Properties

```
#conexão ao postgresql
spring.datasource.url=jdbc:postgresql://localhost:5432/cursodb
spring.datasource.username=postgres
spring.datasource.password=postgres
```

```
#define a forma de geração do banco de dados
spring.jpa.hibernate.ddl-auto=create
```

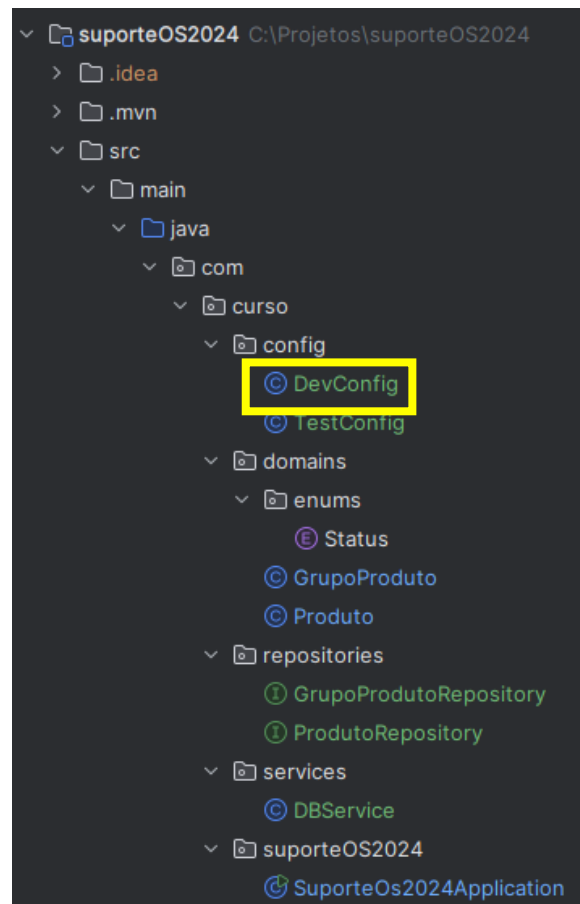
```
# gera log
logging.file.name=suporte0s_dev.log
logging.level.root=INFO
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
```

```
#exibe os SQL gerado no console
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

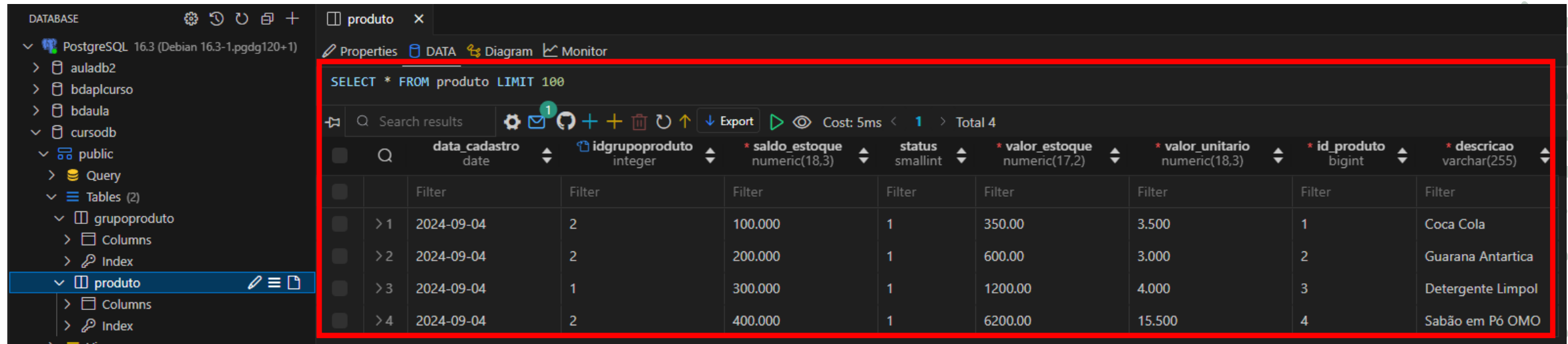
```
1 spring.application.name=suporte0s2024
2 spring.profiles.active=dev
```

Criando Projeto Spring Boot – Conexão BD

- No pacote **config** crie uma nova classe java → **DevConfig.java**.
- Faça o desenvolvimento **de DevConfig.java**:
- Agora através do parâmetro **spring.jpa.hibernate.ddl-auto**, podemos setar se vamos alimentar as tabelas do banco ou não no **PostgreSQL**
- Quando “**create**” ele irá criar as tabelas e alimentá-las com dados iniciais.



```
1 package com.curso.config;
2
3 import com.curso.services.DBService;
4 import jakarta.annotation.PostConstruct;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.context.annotation.Profile;
8
9 @Configuration no usages new *
10 @Profile("dev")
11 public class DevConfig {
12
13     @Autowired 1 usage
14     private DBService dbService;
15
16     @PostConstruct no usages new *
17     public void initDB(){
18         this.dbService.initDB();
19     }
20
21 }
22
```



The screenshot shows a PostgreSQL database client interface. On the left, a sidebar lists the database structure: PostgreSQL 16.3 (Debian 16.3-1.pgdg120+1) with databases 'auladb2', 'bdaplcursor', 'bdaula', and 'cursodb'. Under 'public', there are 'Query', 'Tables (2)', and 'produto' (selected). The main panel shows the 'produto' table with a red border. It displays the SQL query 'SELECT * FROM produto LIMIT 100' and a table of 4 rows. The table columns are: 'data_cadastro' (date), 'idgrupoproduto' (integer), 'saldo_estoque' (numeric(18,3)), 'status' (smallint), 'valor_estoque' (numeric(17,2)), 'valor_unitario' (numeric(18,3)), 'id_produto' (bigint), and 'descricao' (varchar(255)).

		data_cadastro date	idgrupoproduto integer	* saldo_estoque numeric(18,3)	status smallint	* valor_estoque numeric(17,2)	* valor_unitario numeric(18,3)	* id_produto bigint	* descricao varchar(255)
	> 1	2024-09-04	2	100.000	1	350.00	3.500	1	Coca Cola
	> 2	2024-09-04	2	200.000	1	600.00	3.000	2	Guarana Antartica
	> 3	2024-09-04	1	300.000	1	1200.00	4.000	3	Detergente Limpol
	> 4	2024-09-04	2	400.000	1	6200.00	15.500	4	Sabão em Pó OMO

- **Informações inseridas com sucesso**