

# JPA e Spring Boot



O que é o JPA - Java Persistence API - uma biblioteca que armazena e recupera objetos que são armazenados em bancos de dados. Ele é responsável por fazer todas as instruções SQL sem que precisemos escrever uma única linha em nosso código (tudo é gerado nos basidores).

## Setup do projeto

Vamos criar nosso projeto no mesmo modelo de antes, conforme os passos

### Passo 1 - Defina seu grupo e artefato

Spring Initializr Bootstrap your application

Project Maven Project Gradle Project

Language Java Kotlin Groovy

Spring Boot 2.3.0 M3 2.3.0 (SNAPSHOT) 2.2.6 (SNAPSHOT) 2.2.5 2.1.14 (SNAPSHOT) 2.1.13

Project Metadata

- Group: com.generation Defina o Grupo do seu projeto
- Artifact: ecommerce Defina o Artefato do seu projeto

### Passo 2 - Importe as bibliotecas padrão (Spring Web e DevTools)

Dependencies

Options

Dependencies

Search depend... Web, Security, JPA, Actuator, Devtools...

Selected dependencies

- Spring Web
- Spring Boot DevTools

Generate - Ctrl + F Explore - Ctrl + Space Share...

### **Passo 3 - Agora vamos precisar da biblioteca do JPA**

The screenshot shows the 'Dependencies' section of the start.spring.io interface. A red box highlights 'Spring Data JPA' under the search results for 'JPA'. A callout bubble says: 'Agora vamos incluir a dependência Spring Data JPA (Java Persistence API)'. Other listed dependencies include 'Spring Web' and 'Spring Boot DevTools', both with green checkmarks.

### **Passo 4 - Como vamos trabalhar com MySQL, vamos precisar também do Driver**

The screenshot shows the 'Dependencies' section with 'MySQL' selected from the search results. A red box highlights 'MySQL Driver'. A callout bubble says: 'MySQL Driver, responsável pela conexão entre nossa API e o banco MySQL'. Other selected dependencies are 'Spring Web', 'Spring Boot DevTools', and 'Spring Data JPA'.

### **Passo 5 - Nosso projeto tem que ficar com as seguintes dependências**

Agora basta você gerar o arquivo e importar no Eclipse

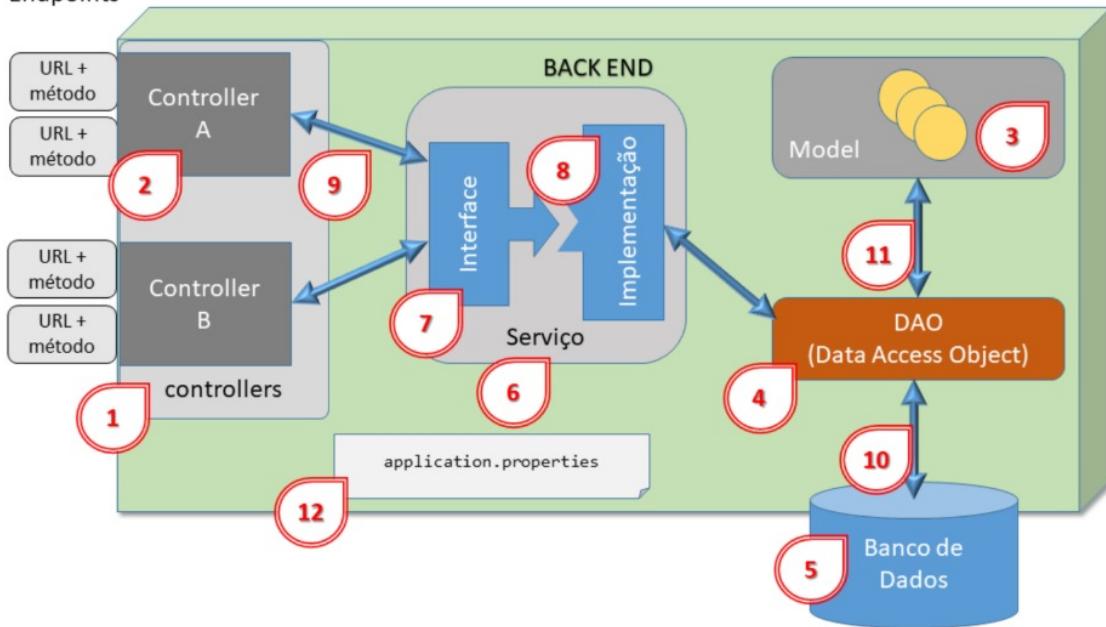
The screenshot shows the 'Dependencies' section with four dependencies selected: 'Spring Web', 'Spring Boot DevTools', 'Spring Data JPA', and 'MySQL Driver'. A red box highlights the 'Generate - Ctrl + ⌘' button at the bottom left. A callout bubble says: 'Temos que garantir que essas 4 dependências estão selecionadas para começar nosso projeto!'. Another callout bubble says: 'Agora é só gerar o projeto e ir para o eclipse'.

**Mais antes...**

## **Entendendo a Arquitetura do Back End**

# ARQUITETURA

Endpoints



1. Precisamos de um pacote de Controllers para oferecer os endpoints para o Front
2. Cada Controller é uma classe específica anotada com `@RestController` composta de métodos que implementam os mapeamentos dos métodos (ex: `GetMapping`, `PostMapping`, `PutMapping`, `DeleteMapping` ).
3. Precisamos dos nossos objetos de negócio. Eles que efetivamente armazenarão as informações sobre o que queremos do nosso sistema.
4. Para armazenar estes objetos de negócio, precisamos implementar uma camada que chamamos de **DAO** (Data Access Object) para que o JPA armazene em tabelas os valores dos atributos dos objetos.
5. Obviamente que nosso banco de dados precisa estar criado. Neste caso, existem 2 alternativas para a implementação das tabelas.
  - Alternativa 1:** Deixar a responsabilidade para o JPA através da configuração `spring.jpa.hibernate.ddl-auto = update` (no arquivo de configuração do projeto - passo 12).
  - Alternativa 2:** não deixar essa responsabilidade para o JPA, porém o programador deve criar as tabelas manualmente.
6. Os Controlles não podem acessar diretamente os objetos do DAO (não é boa prática). Para isso, criamos um novo pacote com classes que implementam essa "conversa". A esse pacote chamamos de **servico**.
7. Todo serviço também tem um modo de ser declarado. Sempre declaramos a interface do serviço com os possíveis métodos que farão essa conversão (ex: invocar a gravação de um objeto no banco, recuperar um único objeto, recuperar vários objetos, etc).

8. Não basta apenas a interface, precisamos da lógica destes métodos. Para isso, criamos objetos que implementam estas interfaces e anotamo-os com `@Component` para indicar que serão injetáveis.

9. A chamada dos serviços é declarada nos controllers através da referência à interface do serviço (anotada com `@Autowired`). O SpringBoot vai sozinho encontrar o objeto que implementa isso. Como ele faz? Buscando os objetos que estão anotados com `@Component`.

10. A comunicação entre DAO e Banco de Dados ocorre através da criação de comandos SQL, porém isso é transparente para o programador (significa que não precisamos escrever nenhuma instrução SQL).

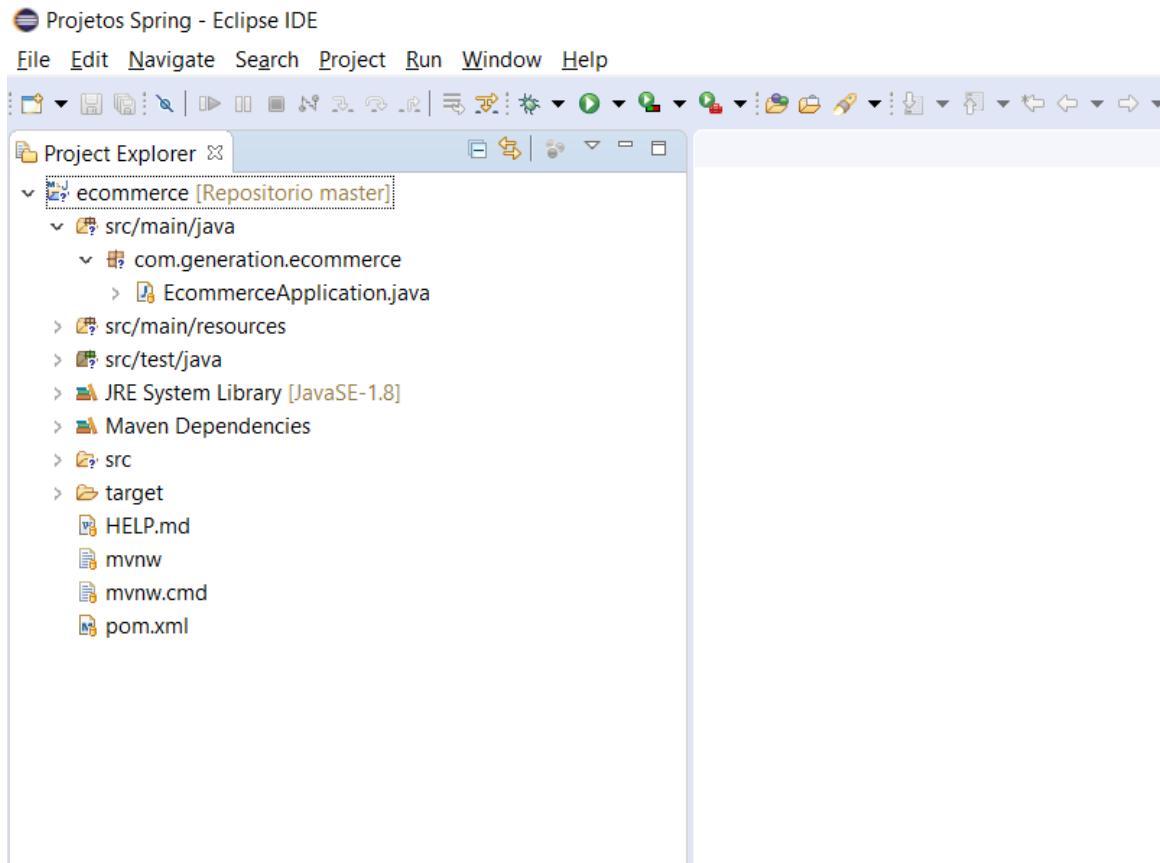
11. Tanto na inserção quanto na recuperação, quem preenche ou consulta os valores dos atributos dos objetos é o próprio JPA.

12. Tudo só é possível, porque existe um arquivo chamado `application.properties` que configura nosso projeto (desde a porta que o BackEnd vai atender, até os parâmetros de conexão com o banco de dados).

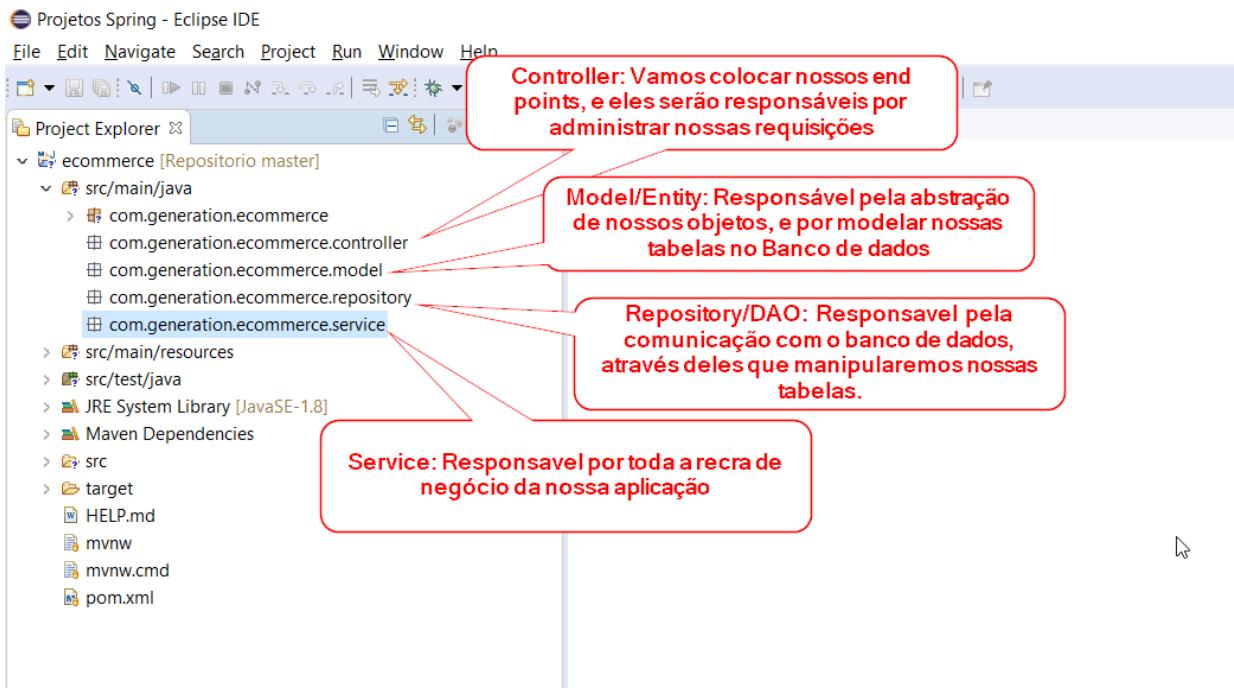
## Criando nosso projeto JPA Spring Boot

---

- Uma vez importado nosso projeto para o Eclipse, teremos:

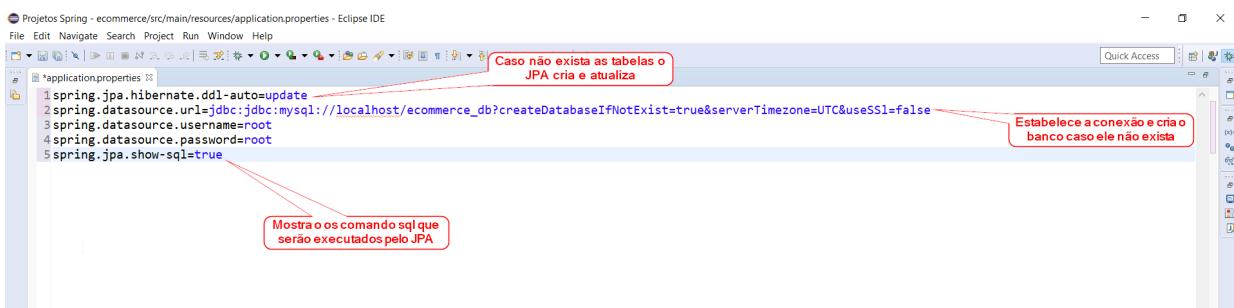


- Agora precisamos criar os nossos pacotes que definirão a arquitetura do nosso projeto.



## Configurando a Conexão com o Banco de dados.

Precisamos estabelecer a nossa conexão com o banco de dados, na pasta resources temos um arquivo chamado `application.properties` é nele que iremos configurar a conexão com o banco de dados.



Neste arquivo precisamos definir alguns parâmetros que definirão a conexão como nossa base de dados.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost/ecommerce_db?
createDatabaseIfNotExist=true&serverTimezone=UTC&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
Admin357/ spring.jpa.show-sql=true
```

- Vamos entender um pouco do código acima:



```

Projeto Spring - ecommerce/src/main/resources/application.properties - Eclipse IDE
File Edit Navigate Search Project Run Window Help
application.properties
1spring.jpa.hibernate.ddl-auto=update
2spring.datasource.url=jdbc:mysql://localhost/ecommerce_db?createDatabaseIfNotExist=true&serverTimezone=UTC&useSSL=false
3spring.datasource.username=root
4spring.datasource.password=root
5spring.jpa.show-sql=true

```

## Requisitos do nosso sistema

---

Nosso e-commerce terá:

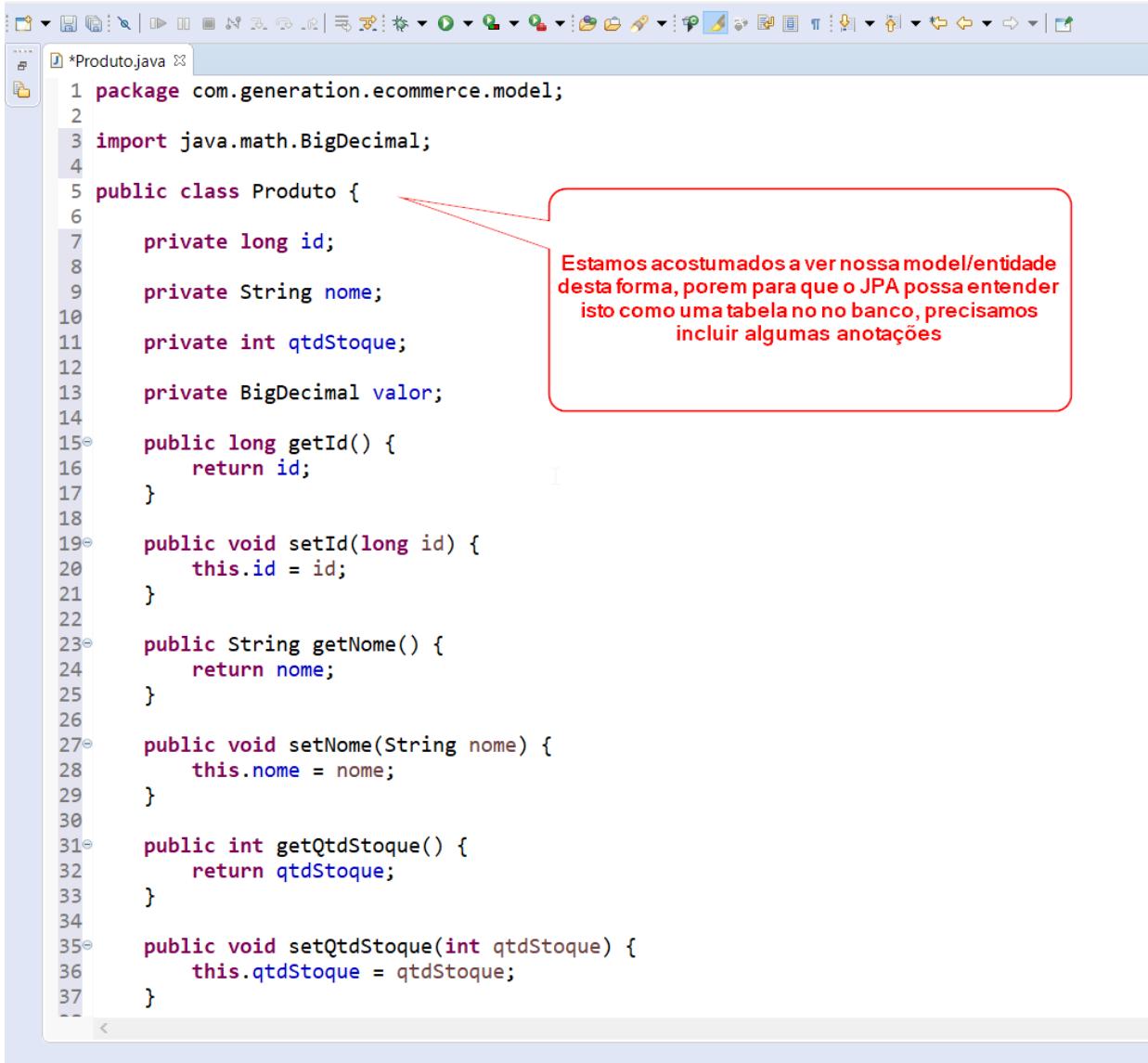
1. CRUD de produtos onde serão cadastrados os produtos, valor, uma foto do produto e quantidade em estoque.
2. CRUD de usuário podendo ser vendedor ou cliente ( indicado por um booleano Boolean vendedor ).
3. CRUD de categoria onde será relacionado diretamente com a Entidade de produto, onde: Uma categoria deverá ter muitos produtos, e Muitos produtos deverá ter apenas uma Categoria por produto.
4. Todos os CRUD's deverão conter as seguintes camadas:
  - Model/Entity
  - Repository/DAO
  - Service (Caso exista alguma regra de negócio)
  - Controller com os Métodos HTTP **Get, Post, Put e Delete.**

## Agora Mão na Massa!

### Crud Produtos

#### 1. Model/Entity

Vamos criar a nossa Entidade Produto.



```

1 package com.generation.e-commerce.model;
2
3 import java.math.BigDecimal;
4
5 public class Produto {
6
7     private long id;
8
9     private String nome;
10
11    private int qtdStoque;
12
13    private BigDecimal valor;
14
15    public long getId() {
16        return id;
17    }
18
19    public void setId(long id) {
20        this.id = id;
21    }
22
23    public String getNome() {
24        return nome;
25    }
26
27    public void setNome(String nome) {
28        this.nome = nome;
29    }
30
31    public int getQtdStoque() {
32        return qtdStoque;
33    }
34
35    public void setQtdStoque(int qtdStoque) {
36        this.qtdStoque = qtdStoque;
37    }

```

Estamos acostumados a ver nossa model/entidade desta forma, porém para que o JPA possa entender isto como uma tabela no banco de dados, precisamos incluir algumas anotações

Agora vamos adicionar todas as anotações que o JPA irá utilizar para mapear a nossa Model/Entity e irá transformar em tabela no nosso banco de dados, nossa Model/Entity ficará assim:

```

package com.generation.e-commerce.model;

import java.math.BigDecimal;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.Min;
import javax.validation.constraints.Size;
import com.sun.istack.NotNull;

@Entity
@Table(name = "produto")

```

```
public class Produto {

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

    @NotNull
    @Size(min = 5, max = 256)
    private String nome;

    @Min(0)
    private int qtdStoque;

    private BigDecimal valor;

    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 int getQtdStoque() {
        return qtdStoque;
    }

    public void setQtdStoque(int qtdStoque) {
        this.qtdStoque = qtdStoque;
    }

    public BigDecimal getValor() {
        return valor;
    }

    public void setValor(BigDecimal valor) {
        this.valor = valor;
    }
}
```

Vamos entender o que significa cada uma dessas anotações:

```
Projeto Spring - ecommerce/src/main/java/com/generation/ecommerce/model/Produto.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
* Produto.java
16 @Entity
17 @Table(name = "produto")
18 public class Produto {
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private long id;
22
23     @NotNull
24     @Size(min = 5, max = 256)
25     private String nome;
26
27     @Min(0)
28     private int qtdStoque;
29
30     private BigDecimal valor;
31
32     public long getId() {
33         return id;
34     }
35
36     public void setId(long id) {
37         this.id = id;
38     }
39
40     public String getNome() {
41         return nome;
42     }
43
44     public void setNome(String nome) {
45         this.nome = nome;
46     }
47
48     public int getQtdStoque() {
49         return qtdStoque;
50     }
51
52 }
```

Annotations explained:

- `@Entity`: Informa que esta Model é uma Entidade e que ela virará uma tabela no banco de dados.
- `@Table(name = "produto")`: Define o nome da tabela no banco de dados como produto.
- `@Id`: Define a propriedade id como uma Primary Key no nosso banco de dados.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Não permite que o client (browser) insira valores nulos para a nossa API. Define os valores automaticamente de forma crescente (Auto\_Increment no banco de dados).
- `@NotNull`: Define uma quantidade mínima e máxima de caracteres.
- `@Size(min = 5, max = 256)`: Define um valor numérico mínimo aceito.
- `@Min(0)`: Define uma quantidade mínima e máxima de caracteres.
- `@Min(0)`: Define um valor numérico mínimo aceito.

Bom! agora que ja temos a nossa Model/Entity pronta, vamos construir a nossa interface de **Repositpory**.

## 2. Repository/DAO

Vamos criar o nossa Interface de Repository

Como ja vimos anteriormente, a interface de repository é responsável pela comunicação com a nossa base de dados, é ela que irá realizar as nossas consultas.

Agora vamos ver como isto fica no código:

```
package com.generation.ecomerce.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.generation.ecomerce.model.Produto;

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

Podemos realizar nossas próprias consultas personalizadas assim basta inserir a nossa consulta através do Query Method:

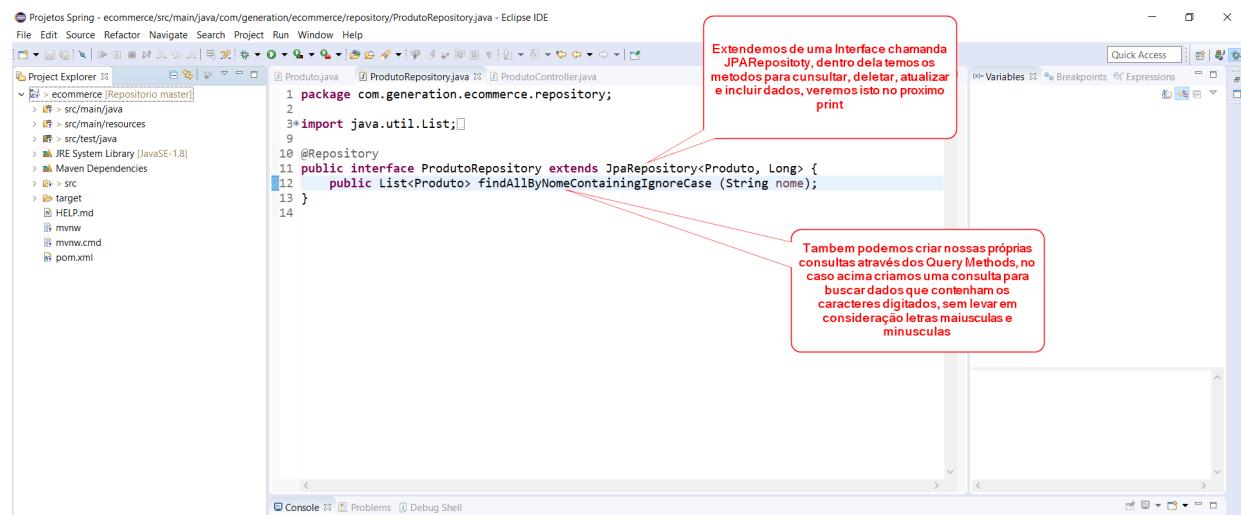
```
public List<Produto> findAllByNomeContainingIgnoreCase (String nome);
```

Nossa classe ficará assim:

```

@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long> {
    public List<Produto> findAllByNomeContainingIgnoreCase (String nome);
}

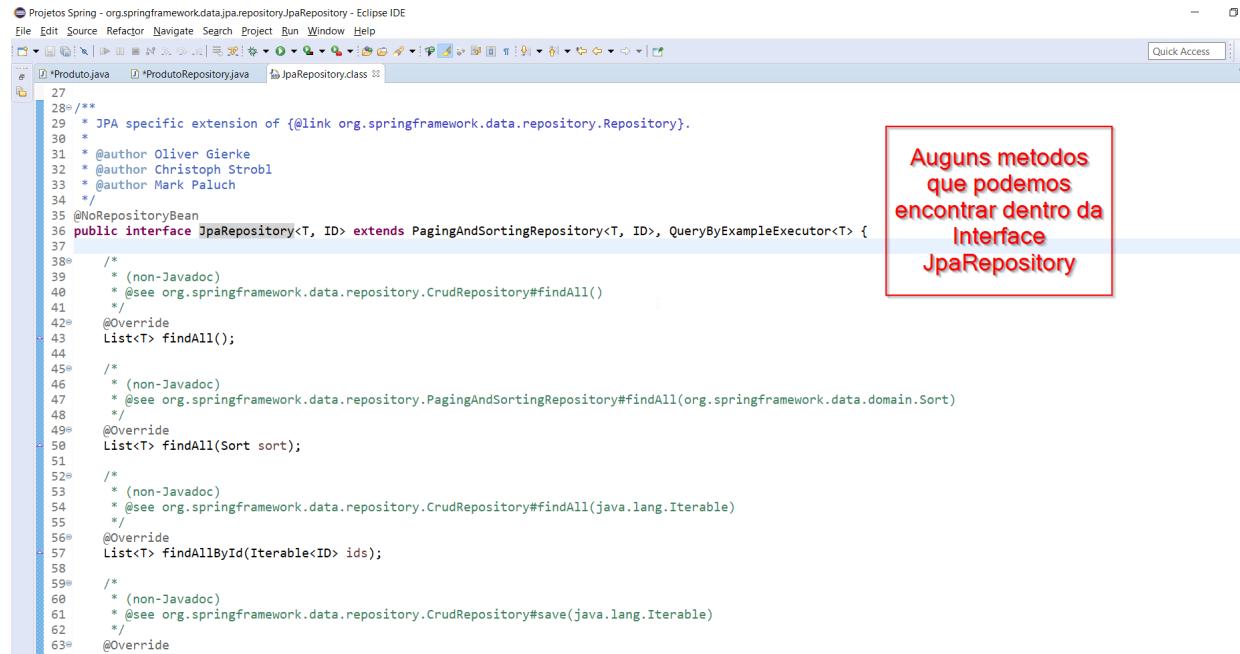
```



Podemos consultar mais parâmetros que podemos acrescentar em nossos Query Methods na documentação do [Spring Data JPA](#)

A Interface JPARepositoty já contém todos os métodos de consulta básica, ou seja métodos como findAll, findById, save, e delete e etc já contem nesta interface.

Vamos acessar a interface e ver estes métodos:



### 3. Controller

Agora precisamos criar o nosso ProdutoController onde faremos o controlos dos nossos endpoits e disponibilizaremos nossos recursos para o client.

Damos o Nome de **Recurso** para as informações devolvidas pela Api, ex: podemos entender que o produto entregue pela Api é um recurso de Produto.

Vamos Montar nossa primeira Classe de Controller.

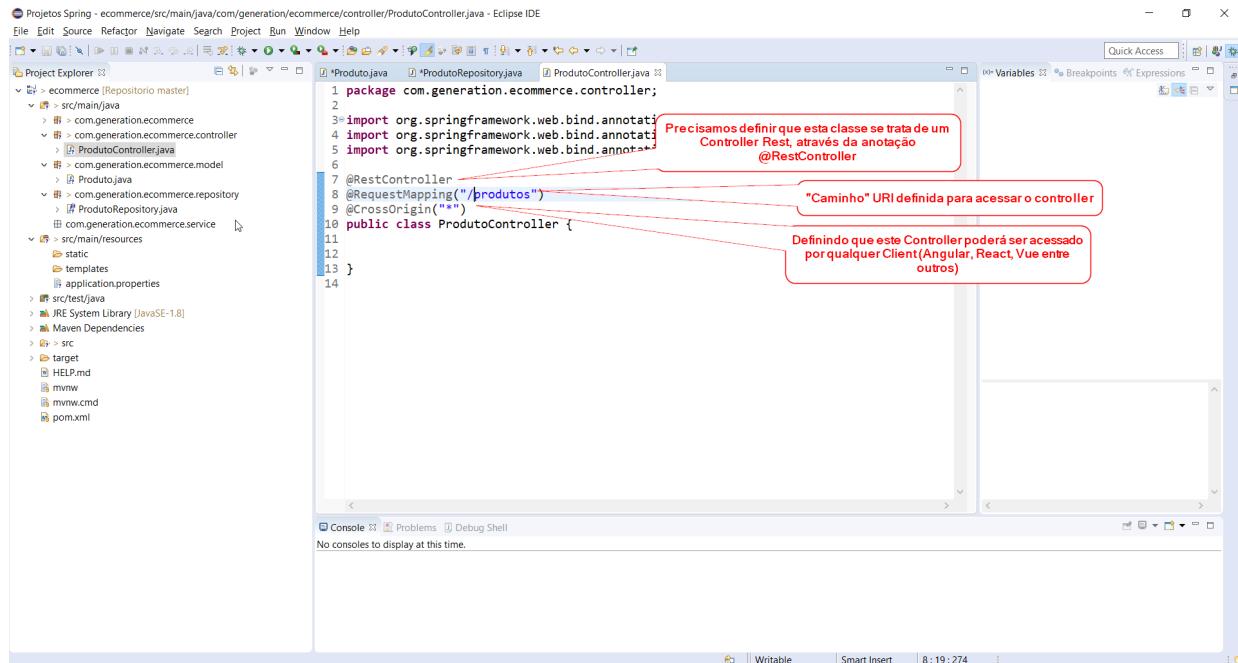


Imagen acima demonstra como configurar um Controller

Se atente com a anotação `@CrossOrigin`, esta anotação é imprescindível para liberar o back-end para um aplicação externa em outra porta.

### 3.1 Vamos fazer a consulta de todos os recursos utilizando o `findAll()`

- O método `findAll()` retorna todos item da nossa tabela.

Crie uma classe controller e coloque o seguinte código

```
@RestController
@RequestMapping( "/produtos" )
@CrossOrigin( "*" )
public class ProdutoController {

    @Autowired
    private ProdutoRepository repository;

    @GetMapping
    public ResponseEntity<List<Produto>> GetAll(){
        return ResponseEntity.ok(repository.findAll());
    }
}
```

Agora vamos entender um pouco do que significa cada código.

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the project structure under "ecommerce [Repositorio master]".
- Code Editor:** Displays the `ProdutoController.java` file content.
- Annotations and Methods:** The code includes annotations like `@RestController`, `@RequestMapping`, and `@GetMapping`. It also defines a private field `private ProdutoRepository repository;` and a method `GetAll()` that returns a `ResponseEntity<List<Produto>>`.
- Annotations Callouts:**
  - A callout points to `@RestController` with the text: "Estamos definindo o metodo como um metodo HTTP-GET".
  - A callout points to `ResponseEntity<...>` with the text: "ResponseEntity<...> é uma interface que monta um template no protocolo HTTP passando um body a lista de produtos em formato Json."
  - A callout points to `repository.findAll()` with the text: "findAll() é um metodo implementado pela Interface JPA... usamos ela para carregar todos os dados na tabela Produtos".
  - A callout points to the `repository` field with the text: "Como não conseguimos instanciar uma interface, precisamos utilizar uma injeção de dependencia (@Autowired) para encarregar o Spring de instanciar tudo o que precisa ser carregado".
- Bottom Status Bar:** Shows "Writable", "Smart Insert", "21:42:704", and other status indicators.

Inicie sua aplicação clicando no arquivo principal assim como a imagem abaixo.

1 - Acesse a classe com o final "Application.java"

2 - Clique com o botão direito do Mouse na parte branca da aba que contém o código.

3 - Clique em Run As

4 - Clique em Java Application

5 - Se o spring conseguir iniciar a sua aplicação, na tela de Console aparecerá que sua aplicação foi "Servida" na porta 8080

Para testar o nosso método findAll do controle podemos acessar o aplicativo postman.

O Postman é uma ferramenta que tem como objetivo testar serviços RESTful (Web APIs) por meio do envio de requisições HTTP e da análise do seu retorno. Com ele é possível consumir facilmente serviços locais e na internet, enviando dados e efetuando testes sobre as respostas das requisições.

Acesse o postman de seu computador e siga as instruções abaixo.

Coloque a uri da rota do seu método find All, defina o verbo http como GET, e aperte o botão send

The screenshot shows the Postman interface with the following annotations:

- A red box highlights the "Method" dropdown set to "GET". A callout points to it with the text "Método da requisição".
- A red box highlights the "URL" input field containing "http://localhost:8080/produtos". A callout points to it with the text "URI da nossa Api".
- A red box highlights the "Body" tab. A callout points to it with the text "URL da nossa Api".
- A red box highlights the status bar at the bottom showing "Status: 200 OK". A callout points to it with the text "Status entregue pela API através do ResponseEntity<>".

Se a requisição for realizada com sucesso aparecerá um status 200. mostrando que o método foi realizado adequadamente.

Agora para analisarmos se o JPA hibernate criou as tabelas de forma adequada podemos checar utilizando o app do MySQL Workbench.

Acesse o mysql workbench de seu computador.

The screenshot shows the MySQL Workbench interface with the following annotations:

- A red box highlights the "ecommerce\_db" schema in the Navigator panel. A callout points to it with the text "O Banco de dados conforme especificamos no application.properties".
- A red box highlights the "produto" table in the Navigator panel. A callout points to it with the text "A tabela produto, conforme especificamos na nossa Entity".
- A red box highlights the "Result Grid" showing an empty table with columns "id", "nome", "qtd\_stoque", and "valor". A callout points to it with the text "A tabela esta vazia, por isso que não recebemos nenhum dado na body".

Repare que foi criado a tabela produto no banco de dados com as colunas/atributos que definimos na model assim como o previsto.

Insira os seguintes dados na tabela produto

| id | nome      | qtd_stoque | valor |
|----|-----------|------------|-------|
| 1  | Radio MP3 | 50         | 30.00 |

Retorne ao postman e execute o método GetAll de produtos mais uma vez para chamarmos o produto que inserimos na tabela.

```
1
2   {
3     "id": 1,
4     "nome": "Radio MP3",
5     "qtdStoque": 50,
6     "valor": 30.00
7   }
```

### 3.2 Vamos fazer a consulta de um recurso utilizando o `findById()`

O método `findById` retorna apenas um item da nossa tabela.

Volte ao Spring para criar o método `findById`, crie o seguinte código abaixo no controller.

**Importante: O código no mesmo arquivo `ProdutoController`, logo abaixo do método `GetAll`.**

```
@GetMapping("/{id}")
public ResponseEntity<Produto> GetById(@PathVariable long id){
    return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
        .orElse(ResponseEntity.notFound().build());
}
```

## Vamos entender o código:

The screenshot shows the Eclipse IDE interface with the code editor open to `ProdutoController.java`. The code defines a `ProdutoController` class with two methods: `list` and  `GetById`. The  `GetById` method uses `@PathVariable` to receive an ID from the URL and returns a `ResponseEntity<Produto>`. A red callout box points to the `@PathVariable` annotation with the text: "Uma forma de passar valores para o nossa API é através de PathVariable, neste caso quando formos chamar metodo devemos passar da seguinte forma: http://localhost:8080/produtos/1". Another red callout box points to the `map` method with the text: "Usando o método map() com expressão lambda para informar que caso venha valor nulo o Response sera de um 404 not found". A third red callout box points to the `repository.findById(id)` line with the text: "Sempre quando recebemos um valor via PathVariable em nossos metodos, precisamos anotar a variável que receberá o valor com @PathVariable". The console tab at the bottom shows application logs.

```
Projeto Spring - ecommerce/src/main/java/com/generation/e-commerce/controller/ProdutoController.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer [ecommerce] (Repositorio master)
src/main/java
  com.generation.e-commerce
    EcommerceApplication.java
  com.generation.e-commerce.controller
    ProdutoController.java
  com.generation.e-commerce.model
    Produto.java
  com.generation.e-commerce.repository
    ProdutoRepository.java
  com.generation.e-commerce.service
src/main/resources
  static
  templates
  application.properties
src/test/java
Maven Dependencies
src
target
  maven
  mvnw
  mvnw.cmd
pom.xml

Console [Problems] [Debug Shell]
EcommerceApplication [Java Application] [JavaSE-1.8]
2020-03-13 14:42:05.394 INFO 19232 --- [ restartedMain] .ConditionEvaluationBean : ConditionEvaluationBean initialized successfully with 0 conditions evaluated
2020-03-13 14:42:16.763 INFO 19232 --- [nio-8080-exec-9] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-03-13 14:42:16.763 INFO 19232 --- [nio-8080-exec-9] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-03-13 14:42:16.769 INFO 19232 --- [nio-8080-exec-9] o.s.web.servlet.DispatcherServlet : Completed initialization
Hibernate: select produto0_.id as id1_0_0_, produto0_.nome as nome2_0_0_, produto0_.qtd_stoque as qtd_stoq3_0_0_, produto0_.va
Hibernate: select produto0_.id as id1_0_0_, produto0_.nome as nome2_0_0_, produto0_.qtd_stoque as qtd_stoq3_0_0_, produto0_.va
2020-03-13 14:42:05.394 INFO 19232 --- [ restartedMain] .ConditionEvaluationBean : ConditionEvaluationBean initialized successfully with 0 conditions evaluated
2020-03-13 14:42:16.763 INFO 19232 --- [nio-8080-exec-9] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-03-13 14:42:16.763 INFO 19232 --- [nio-8080-exec-9] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-03-13 14:42:16.769 INFO 19232 --- [nio-8080-exec-9] o.s.web.servlet.DispatcherServlet : Completed initialization
Hibernate: select produto0_.id as id1_0_0_, produto0_.nome as nome2_0_0_, produto0_.qtd_stoque as qtd_stoq3_0_0_, produto0_.va
Hibernate: select produto0_.id as id1_0_0_, produto0_.nome as nome2_0_0_, produto0_.qtd_stoque as qtd_stoq3_0_0_, produto0_.va
Iniciar
```

Repare na anotação `@PathVariable`, esta anotação utilizada quando o valor da variável é passada diretamente na URL, ou seja no nosso contexto o ID, que usaremos para definir qual produto que queremos buscar no nosso banco de dados.

Feito isso vamos testar este método no postman.

The screenshot shows the Postman interface with a new request named "Untitled Request". The method is set to GET and the URL is `http://localhost:8080/produtos/1`. In the "Params" tab, there is a single parameter "id" with the value "1". The response body is displayed in JSON format, showing the details of the product with ID 1: { "id": 1, "name": "Radio MP3", "qtdStoque": 50, "valor": 30.00 }. The status bar at the bottom indicates a 200 OK response.

Repare que foi possível encontrar um produto com todas as informações que nós definimos na model, produto que adicionamos no MySQL Workbench, a requisição é dada com sucesso quando ela retorna um status 200 como podemos perceber na imagem.

## Exemplo de requisição mandando um ID inexistente na tabela do banco de dados.

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'History', 'Collections' (which is selected), 'APIs', and 'Trash'. Under 'Collections', there are two items: 'Consultas CMKPMoney' with 2 requests and 'WSFSAS' with 5 requests. The main workspace is titled 'Untitled Request' and shows a GET request to 'http://localhost:8080/produtos/155'. A red callout box points to the URL with the text 'Passando um id que não existe na nossa base'. Below the request, the 'Body' tab is selected, showing a single entry '1'. The status bar at the bottom right indicates 'Status: 404 Not Found'.

Repare que o nosso método responde com status 404 not found, mostrando que não foi possível encontrar um produto com aquele ID

Para buscarmos um produto por nome precisamos de um método de busca por nome `getByName()`

3.3 Vamos fazer a consulta por nome de recurso utilizando o `getByName()`, Isto é bem simples, basta utilizar o Query Method que fizemos no Repository em no endPoint em nosso controller, ficará assim:

```
@GetMapping("/nome/{nome}")
public ResponseEntity<List<Produto>> GetByNome(@PathVariable String nome)
    return ResponseEntity.ok(repository.findAllByNameContainingIgnoreCase(nome))
}
```

## Vamos entender o código:

```
1 package com.generation.ecommerce.controller;
2
3 import java.util.List;
4
5 @RestController
6 @RequestMapping("/produtos")
7 @CrossOrigin("*")
8 public class ProdutoController {
9
10     @Autowired
11     private ProdutoRepository repository;
12
13     @GetMapping
14     public ResponseEntity<List<Produto>> GetAll(){
15         return ResponseEntity.ok(repository.findAll());
16     }
17
18     @GetMapping("/{id}")
19     public ResponseEntity<Produto> GetById(@PathVariable Long id){
20         return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
21             .orElse(ResponseEntity.notFound().build());
22     }
23
24     @GetMapping("/nome/{nome}")
25     public ResponseEntity<List<Produto>> GetByNome(@PathVariable String nome){
26         return ResponseEntity.ok(repository.findAllByNameContainingIgnoreCase(nome));
27     }
28 }
```

A red callout box points to the line `@GetMapping("/{id}")` with the text: "Temos que criar uma nova rota para acessarmos o recurso desejado". Another red callout box points to the line `@GetMapping("/nome/{nome}")` with the text: "Você se lembram da nossa consulta personalizada MethodQuery que fizemos no ProdutoRepository, estamos chamando ela neste endPoint. Vamos testar?".

## Exemplo de teste pelo postman.

The Postman interface shows a GET request to `http://localhost:8080/produtos/nome/Ra`. A red callout box points to the URL with the text: "Nosso caminho personalizado". The response status is 200 OK, and the response body is a JSON object:

```
1 {
2     "id": 1,
3     "name": "Radio MP3",
4     "qtdStock": 50,
5     "valor": 30.00
6 }
```

Red callout boxes highlight the success status message "Resposta Indicando o Sucesso na requisição" and the returned JSON data "Recurso devolvido na Body".

Ao pesquisarmos um valor que não existe na nossa base de dados receberemos apenas um array vazio, não sendo necessário realizar uma tratativa de erro

| KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|-----|-------|-------------|-----|-----------|
| Key | Value | Description |     |           |

1 [{}]

Array Vazio

Para inserirmos um produto na tabela de produto nosso banco de dados precisamos de um método método de post por nome post()

### 3.4 Vamos fazer post de recurso utilizando o post()

```
@PostMapping
public ResponseEntity<Produto> Post(@RequestBody Produto produto){
    return ResponseEntity.ok(repository.save(produto));
}
```

## Vamos entender o código:

```
1 package com.generation.ecommerce.controller;
2
3 import java.util.List;
4
5 @RestController
6 @RequestMapping("/produtos")
7 @CrossOrigin("*")
8 public class ProdutoController {
9
10     @Autowired
11     private ProdutoRepository repository;
12
13     @GetMapping
14     public ResponseEntity<List<Produto>> GetAll(){
15         return ResponseEntity.ok(repository.findAll());
16     }
17
18     @GetMapping("/{id}")
19     public ResponseEntity<Produto> GetById(@PathVariable long id){
20         return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
21             .orElse(ResponseEntity.notFound().build());
22     }
23
24     @GetMapping("/{nome}")
25     public ResponseEntity<List<Produto>> GetByName(@PathVariable String nome){
26         return ResponseEntity.ok(repository.findAllByName(nome));
27     }
28
29     @PostMapping
30     public ResponseEntity<Produto> Post(@RequestBody Produto produto){
31         return ResponseEntity.ok(repository.save(produto));
32     }
33
34 }
```

Diferente do método acima, precisamos passar um objeto ao invés de um valor único, por isto devemos passar no body da requisição e para recepcionar este objeto na nossa api precisamos utilizar a Anotação `@RequestBody` em nosso Objeto do tipo produto.

Este endPoint só será acessado se na requisição que estiver com o método postem seu cabeçalho

Vamos usar o método `save()` para salvar o produto novo em nossa base de dados e também vamos de volver o mesmo produto no body da resposta da nossa requisição

Respare que para esse tipo de requisição post precisamos de uma anotação de `@PostMapping`

Repare na anotação `@PathVariable`, esta anotação mapeia o corpo `HttpRequest` para um objeto de transferência ou domínio, permitindo a desserialização automática do corpo `HttpRequest` de entrada em um objeto Java.

Para testarmos via postman precisamos enviar a url contrarrespondente ao método post enviar a informação do objeto produto no corpo da requisição e selecionar o verbo post no botão ao lado da caixa de seleção da URL, como mostra a imagem a baixo

Passando o Método Post no cabeçalho da requisição

Chamaremos a URI Padrão pois não definimos um "sub caminho" na nossa API

Clique na Body para enviar o nosso Objeto

Beautify

Selezione o formato Json

Clique em raw

O objeto em formato Json que enviaremos na nossa requisição

Ao clicarmos em "Send", se todos os dados estiverem corretos, receberemos um Status 200 OK e o objeto já com o id que foi cadastrado na nossa base de dados

Repare que não enviamos id no corpo, com isso garantimos que podemos incluir um novo produto.

Veja o dado inserido no banco através do MySQL Workbench.

MySQL Workbench

Local instance MySQL X

File Edit View Query Database Server Tools Scripting Help

Navigator: Local instance MySQL X

SCHEMAS: ecommerce\_db

Table: produto

Columns:

| id | nome       | qtd_stoque | valor   |
|----|------------|------------|---------|
| 1  | Radio MP3  | 50         | 30.00   |
| 2  | Video Game | 30         | 1200.00 |

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: |

Output: Action Output | Time Action | 1 11:00:38 SELECT \* FROM ecommerce\_db.produto LIMIT 0, 1000 | Message: 2 row(s) returned | Duration / Fetch: 0.000 sec / 0.000 sec

Object Info Session

Alteração feita no Banco de Dados

Para atualizarmos um produto na tabela de produto nosso baco de dados precisamos de um método método de Put(), ele será responsável por localizar o item e atualizar.

### 3.4 Vamos fazer uma atualização de recurso utilizando o Put()

```

@PutMapping
public ResponseEntity<Produto> Put(@RequestBody Produto produto){
    return ResponseEntity.ok(repository.save(produto));
}

```

## Vamos entender o código

```

19 @RestController
20 @RequestMapping("/produtos")
21 @CrossOrigin("*")
22 public class ProdutoController {
23
24    @Autowired
25    private ProdutoRepository repository;
26
27    @GetMapping
28    public ResponseEntity<List<Produto>> GetAll(){
29        return ResponseEntity.ok(repository.findAll());
30    }
31
32    @GetMapping("/{id}")
33    public ResponseEntity<Produto> GetById(@PathVariable long id){
34        return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
35            .orElse(ResponseEntity.notFound().build());
36    }
37
38    @GetMapping("/nome/{nome}")
39    public ResponseEntity<List<Produto>> GetByNome(@PathVariable String nome){
40        return ResponseEntity.ok(repository.findAllByName(nome));
41    }
42
43    @PutMapping
44    public ResponseEntity<Produto> Put(@RequestBody Produto produto){
45        return ResponseEntity.ok(repository.save(produto));
46    }
47
48    @PutMapping
49    public ResponseEntity<Produto> Put(@RequestBody Produto produto){
50        return ResponseEntity.ok(repository.save(produto));
51    }
52
53
54 }

```

**Annotations and Logic:**

- @RestController**: Marks the class as a REST controller.
- @RequestMapping("/produtos")**: Maps the controller to the "/produtos" endpoint.
- @CrossOrigin("\*")**: Allows requests from any origin.
- @GetMapping**: Maps the `GetAll()` method to the root endpoint.
- @GetMapping("/{id})**: Maps the `.GetById()` method to the endpoint with path variable `{id}`.
- @GetMapping("/nome/{nome})**: Maps the `GetByNome()` method to the endpoint with path variable `{nome}`.
- @PutMapping**: Maps the `Put(@RequestBody Produto produto)` methods to handle PUT requests.

**Annotations with Red Boxes:**

- Line 43:** `@PutMapping` with annotation `@RequestBody Produto produto`. A red box surrounds this line with the text: "Através do Método Put podemos fazer a atualização de um dado específico em nossa base de dados."
- Line 44:** `@PutMapping` with annotation `@RequestBody Produto produto`. A red box surrounds this line with the text: "Receberemos o objeto da mesma forma que no endPoint de Post"
- Line 48:** `@PutMapping` with annotation `@RequestBody Produto produto`. A red box surrounds this line with the text: "Chamaremos o mesmo método save(), este método serve para ambos as finalidades (Salvar e Atualizar). Quando enviamos um objeto sem o Id o JPA entende isto como um Poste quando enviamos o objeto com o atributo Id populado o JPA faz um find através do Id passado junto com o objeto e atualiza caso haja retorno"

Repare que para atualizar o método do Controller se mantém com apenas a modificação da anotação mapping de `@PostMapping` para `@PutMapping`.

Para atualizarmos via post precisamos enviar a url correspondente ao método PUT, enviar a informação do objeto produto no corpo da requisição e selecionar o verbo PUT no botão ao lado da caixa de seleção da URL, como mostra a imagem a baixo

The screenshot shows the Postman interface. At the top, there's a red callout pointing to the 'Headers' section with the text 'Passando no cabeçalho o Método Put'. Below it, the 'Body' tab is selected, showing a JSON object with an 'id' field set to 2. A red callout from this section points to the JSON code with the text 'Objeto com o id populado e a alteração feita, sendo enviado via Body'. At the bottom, the response status is shown as 'Status: 200 OK'.

```

1 - {
2   "id": 2,
3   "nome": "Video Game",
4   "qtdStoque": 30,
5   "valor": 1600.00
6 }

```

Repare que agora precisamos enviar o id no corpo, com isso garantimos que podemos atualizar um produto.

Veja o dado inserido no banco através do MySQL Workbench.

The screenshot shows the MySQL Workbench interface. In the central pane, a table named 'produto' is displayed with two rows. The second row has its 'valor' column updated to 1600.00. A red callout from the previous screenshot points to this value with the text 'Valor alterado na nossa base de dados.'.

|   | id | nome       | qtd_stoque | valor   |
|---|----|------------|------------|---------|
| 1 | 1  | Radio MP3  | 50         | 30.00   |
| 2 | 2  | Video Game | 30         | 1600.00 |

Para deletarmos um produto na tabela de produto no nosso banco de dados precisamos de um método método de Delete()

3.5 Vamos fazer uma atualização de recurso utilizando o Delete()

```

    @DeleteMapping("/{id}")
    public void Delete(@PathVariable long id) {
        repository.deleteById(id);
    }

```

**Importante: O método delete é do tipo void(), ou seja não retorna nada!**

Vamos entender o código:

```

    24
    25*     @Autowired
    26     private ProdutoRepository repository;
    27
    28*     @GetMapping
    29     public ResponseEntity<List<Produto>> GetAll(){
    30         return ResponseEntity.ok(repository.findAll());
    31     }
    32
    33*     @GetMapping("/{id}")
    34     public ResponseEntity<Produto> GetById(@PathVariable long id){
    35         return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
    36             .orElse(ResponseEntity.notFound().build());
    37     }
    38
    39*     @GetMapping("/name/{name}")
    40     public ResponseEntity<List<Produto>> GetByName(@PathVariable String name){
    41         return ResponseEntity.ok(repository.findAllByNameContainingIgnoreCase(name));
    42     }
    43
    44*     @PostMapping("/name/{name}")
    45     public ResponseEntity<Produto> Post(@RequestBody Produto produto){
    46         return ResponseEntity.ok(repository.save(produto));
    47     }
    48
    49*     @PutMapping("/name/{name}")
    50     public ResponseEntity<Produto> Put(@RequestBody Produto produto){
    51         return ResponseEntity.ok(repository.save(produto));
    52     }
    53
    54*     @DeleteMapping("/{id}")
    55     public void Delete(@PathVariable long id) {
    56         repository.deleteById(id);
    57     }
    58
    59 }

```

Precisamos passar o método Delete no cabeçalho para chamarmos este endPoint

Para o delete não precisamos de retornar um objeto do tipo de ResponseEntity<>, neste caso basta retornar void.

Vamos chamar o método deleteById passando o id do produto que queremos deletar

repare que para isso precisamo passar o id como referencia via @pathVariable

Vejamos como podemos testar via postman e conferir a deleção do dado no banco via MySQL Workbench.

Passando o Método Delete no Cabeçalho

Passando o Id do produto que queremos deletar

E como especificamos a api não devolve nada na body

o Status 200 Informando que tudo deu certo

The screenshot shows the MySQL Workbench interface. In the top-left, the Navigator pane displays the schema structure, including the 'ecommerce\_db' database and its 'produto' table. A query window titled 'Query 1' contains the SQL command: 'SELECT \* FROM ecommerce\_db.produto;'. The Result Grid shows two rows of data: one for a product with ID 1 and another for a product with ID 2. A red callout box points to the first row, which has been deleted, with the text 'O dado com o id 1 foi deletado da base de dados'. Below the grid, the 'Output' tab shows the execution history of the query, indicating the deletion of the row with ID 1.

Podemos ver o primeiro passo do projeto completo no [github da Generation](#)

## Crud Usuário

Desafio: agora é sua vez vamos criar um CRUD completo do usuário a partir da model Usuário.

**Importante** : deixaremos todos os prints e o link no [github Generation](#) para você utilizar como material de apoio para esse desafio.

Veja o código da model abaixo.

### 1. Código Model

```
package com.generation.ecommerce.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
@Table(name = "usuarios")
public class Usuario {
```

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private long id;  
  
@NotNull  
@Size(min = 5, max = 100)  
private String nome;  
  
@NotNull  
@Size(min = 5, max = 10)  
private String usuario;  
  
@NotNull  
@Size(min = 5)  
private String senha;  
  
@NotNull  
private boolean vendedor;  
  
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 getUsuario() {  
    return usuario;  
}  
  
public void setUsuario(String usuario) {  
    this.usuario = usuario;  
}  
  
public String getSenha() {  
    return senha;  
}
```

```

public void setSenha(String senha) {
    this.senha = senha;
}

public boolean isVendedor() {
    return vendedor;
}

public void setVendedor(boolean vendedor) {
    this.vendedor = vendedor;
}
}

```

Projetos Spring - ecommerce/src/main/java/com/generation/ecommerce/model/Usuario.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

```

10
11 @Entity
12 @Table(name = "usuarios")
13 public class Usuario {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private long id;
18
19     @NotNull
20     @Size(min = 5, max = 100)
21     private String nome;
22
23     @NotNull
24     @Size(min = 5, max = 10)
25     private String usuario;
26
27     @NotNull
28     @Size(min = 5)
29     private String senha;
30
31     @NotNull
32     private boolean vendedor;
33
34     public long getId() {
35         return id;
36     }
37
38     public void setId(long id) {
39         this.id = id;
40     }
41
42     public String getNome() {
43         return nome;
44     }
45
46     public void setNome(String nome) {

```

Model Usuário

## 2. Código Repository

```

package com.generation.ecommerce.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.generation.ecommerce.model.Usuario;

```

```

public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
    public List<Usuario> findAllByNomeContainingIgnoreCase (String nome);
}

```

```

Projeto Spring - ecommerce/src/main/java/com/generation/ecommerce/repository/UsuarioRepository.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
... Usuario.java UsuarioRepository.java ProdutoRepository.java
1 package com.generation.ecomerce.repository;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6
7 import com.generation.ecomerce.model.Usuario;
8
9 public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
10     public List<Usuario> findAllByNomeContainingIgnoreCase (String nome);
11 }
12

```

### 3. Usuário Controller

```

package com.generation.ecomerce.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.generation.ecomerce.model.Usuario;
import com.generation.ecomerce.repository.UsuarioRepository;

@RestController
@RequestMapping("/usuarios")
@CrossOrigin("*")
public class UsuarioController {

    @Autowired
    private UsuarioRepository repository;
}

```

```
@GetMapping
public ResponseEntity<List<Usuario>> GetAll(){
    return ResponseEntity.ok(repository.findAll());
}

@GetMapping("/{id}")
public ResponseEntity<Usuario> GetById(@PathVariable long id){
    return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
        .orElse(ResponseEntity.notFound().build());
}

@GetMapping("/nome/{nome}")
public ResponseEntity<List<Usuario>> GetByNome(@PathVariable String nome){
    return ResponseEntity.ok(repository.findAllByNomeContainingIgnoreCase(nome));
}

@PostMapping
public ResponseEntity<Usuario> Post(@RequestBody Usuario usuario){
    return ResponseEntity.ok(repository.save(usuario));
}

@PutMapping
public ResponseEntity<Usuario> Put(@RequestBody Usuario usuario){
    return ResponseEntity.ok(repository.save(usuario));
}

@DeleteMapping("/{id}")
public void Delete(@PathVariable long id) {
    repository.deleteById(id);
}
```

```

Projeto Spring - ecommerce/src/main/java/com/generation/ecommerce/controller/UsuarioController.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
UsarioController.java ProdutoController.java
22 @CrossOrigin("/*")
23 public class UsuarioController {
24
25     @Autowired
26     private UsuarioRepository repository;
27
28     @GetMapping
29     public ResponseEntity<List<Usuario>> GetAll(){
30         return ResponseEntity.ok(repository.findAll());
31     }
32
33     @GetMapping("/{id}")
34     public ResponseEntity<Usuario> GetById(@PathVariable long id){
35         return repository.findById(id).map(resp -> ResponseEntity.ok(resp))
36             .orElse(ResponseEntity.notFound().build());
37     }
38
39     @GetMapping("/nome/{nome}")
40     public ResponseEntity<List<Usuario>> GetByNome(@PathVariable String nome){
41         return ResponseEntity.ok(repository.findAllByNameContainingIgnoreCase(nome));
42     }
43
44     @PostMapping
45     public ResponseEntity<Usuario> Post(@RequestBody Usuario usuario){
46         return ResponseEntity.ok(repository.save(usuario));
47     }
48
49     @PutMapping
50     public ResponseEntity<Usuario> Put(@RequestBody Usuario usuario){
51         return ResponseEntity.ok(repository.save(usuario));
52     }
53
54     @DeleteMapping("/{id}")
55     public void Delete(@PathVariable long id) {
56         repository.deleteById(id);
57     }
58 }

```

## Crud Categoria

Agora faremos o nosso serviço de produtos do nosso back-end, o serviço é um serviço crucial para o nosso projeto de e-commerce, ele será responsável pela organização dos nossos produtos.

O nosso código categoria estará relacionada com produtos em referente de **um para muitos (1... N)** isto será muito parecido com o relacionamento entre tabelas que aprendemos em banco de Dados, estou passando para o JPA Hibernate que em **Uma** Categoria pode ter **Muitos** Produtos, para isso precisaremos de duas anotações `@OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)` e a notação `@JsonIgnoreProperties("categoria")`

Então vamos criar nossa Entity e ver como vai ficar.

### 1. Model Categoria

```

package com.generation.ecommerce.model;

import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;

```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity
@Table(name = "categoria")
public class Categoria {

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

    @NotNull
    @Size(min = 2, max = 15)
    private String descricao;

    @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)
    @JsonIgnoreProperties("categoria")
    private List<Produto> produtos;

    public long getId() {
        return id;
    }

    public List<Produto> getProdutos() {
        return produtos;
    }

    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }

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

    public String getDescricao() {
        return descricao;
    }

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

```
    }  
}
```

Vamos entender um pouco mais sobre estas duas anotações.

```
10 import javax.persistence.OneToMany;  
11 import javax.persistence.Table;  
12 import javax.validation.constraints.  
13 import javax.validation.constraints.  
14  
15 import com.fasterxml.jackson.annotation.  
16  
17 @Entity  
18 @Table(name = "categoria")  
19 public class Categoria {  
20  
21     @Id  
22     @GeneratedValue(strategy = GenerationType.IDENTITY)  
23     private long id;  
24  
25     @NotNull  
26     @Size(min = 3, max = 15)  
27     private String descricao;  
28  
29     @OneToMany(mappedBy = "categoria", cascade = CascadeType.ALL)  
30     @JsonIgnoreProperties("categoria")  
31     private List<Produto> produtos;  
32  
33     public long getId() {  
34         return id;  
35     }  
36  
37     public List<Produto> getProdutos() {  
38         return produtos;  
39     }  
40  
41     public void setProdutos(List<Produto> produtos) {  
42         this.produtos = produtos;  
43     }  
44  
45     public void setId(long id) {  
46         this.id = id;  
47     }  
48 }
```

Anotação que indica que Uma categoria terá muitos Produtos

O parâmetro mappedBy indica que devemos mapear a Atributo categoria na Entity Produto (Veremos ele na proxima imagem)

Qualquer alteração que for realizada a uma categoria acarreta em mudanças para os produtos que pertencem a ela. Ex. Se deletarmos uma categoria, todos os produtos dessa mesma categoria serão apagado!

Como receberemos uma coleção de Produtos precisaremos armazena-los em uma Lista do tipo Produtos

Writable

Sma

## 1.2. Model Produtos

**Temos realizar uma pequena alteração na nossa Entity de Produto.**

Precisamos relacionar a nossa Entity Produtos com a Categoria, mas desta vez iremos informar que **Muitos** Produtos poderão termo apenas **Uma** Categoria chamamos isso de **muitos para um (N... 1)**, para que o JPA entenda isso, temos que criar um atributo do tipo categoria em nossa classe de Produto e anota-la com as anotações `@ManyToOne` e `@JsonIgnoreProperties("produto")`, o atributo ficará assim:

```
@ManyToOne  
JsonIgnoreProperties("produtos")  
private Categoria categoria;
```

Vamos entender um pouco mais deste código

```

16
17
18 @Entity
19 @Table(name = "produto")
20 public class Produto {
21
22     @Id
23     @GeneratedValue(stra
24     private long id;
25
26     @NotNull
27     @Size(min = 5, max =
28     private String nome;
29
30     @Min(0)
31     private int qtdStoque;
32
33     @ManyToOne
34     @JsonIgnoreProperties("produto")
35     private Categoria categoria;
36
37     public Categoria getCategoria() {
38         return categoria;
39     }
40
41     public void setCategoria(Categoria categoria) {
42         this.categoria = categoria;
43     }
44
45     private BigDecimal valor;
46
47     public long getId() {
48         return id;
49     }
50
51     public void setId(long id) {
52         this.id = id;

```

Estamos informando para o JPA que muitos produtos podem ter apenas um tipo de Categoria, logo eu posso uma tabela com muitos produtos, porém cada um destes produtos poderá pertencer a apenas uma Categoria.

Quando a nossa API devolver este recurso no formato Json, a Anotação `@JsonIgnoreProperties` não permitirá que ao informar a categoria do produto em questão informe também a lista de produtos que a categoria contém, evitando assim uma recursividade infinita

Receberemos um Objeto do tipo Categoria como um atributo categoria dentro da Classe Produto

## 2. Repository Categoria

Precisamos criar a nossa camada que se comunicará diretamente com a nossa base de dados## vamos chama-la de `CategoriaRepository`.

O Código é feito da mesma forma que fizemos com os Repositórios de Usuário e Produto

**Importante**, não se esqueça de passar como parâmetro do `JpaRepository` a Entidade Categoria.

O código ficará assim:

```

package com.generation.ecommerce.repository;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

import com.generation.ecommerce.model.Categoria;

public interface CategoriaRepository extends JpaRepository<Categoria, Long>
    public List<Categoria> findAllByDescricaoContainingIgnoreCase (String d
}

```

```

1 package com.generation.ecommerce.repository;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6
7 import com.generation.ecommerce.model.Categoria;
8
9 public interface CategoriaRepository extends JpaRepository<Categoria, Long>{
10     public List<Categoria> findAllByDescricaoContainingIgnoreCase (String descricao);
11 }
12

```

### 3. Controller Categorias

Agora precisamos criar a nossa classe que controla-rá nossos }

EndPoints e fornecerá todos os recursos necessários para o client (Aplicação que consumirá nossa API).

Vamos criar mais uma classe com o nome de `ControllerCategoria` a

A esta “Altura do campeonato” ja conseguimos compreender o que tudo que será necessário termos em nosso controller para possamos chamado de **crud (Get, GetAll, Post, Put e Delete)**, então vamos para o código!

```

package com.generation.ecommerce.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin("*")
@RequestMapping("/categorias")
public class CategoriaController {

    @Autowired

```

```
private CategoriaRepository repository;

@GetMapping
public ResponseEntity<List<Categoria>> GetAll() {
    return ResponseEntity.ok(repository.findAll());
}

@GetMapping("/{id}")
public ResponseEntity<Categoria> GetById(@PathVariable long id) {
    return repository.findById(id).map(resp -> ResponseEntity.ok(resp));
}

@GetMapping( "/nome/{nome}" )
public ResponseEntity<List<Categoria>> GetByNome(@PathVariable String n
    return ResponseEntity.ok(repository.findAllByDescricaoContainingIgn
}

@PostMapping
public ResponseEntity<Categoria> Post(@RequestBody Categoria categoria)
    return ResponseEntity.ok(repository.save(categoria));
}

@PutMapping
public ResponseEntity<Categoria> Put(@RequestBody Categoria categoria)
    return ResponseEntity.ok(repository.save(categoria));
}

@DeleteMapping( "/{id}" )
public void Delete(@PathVariable long id) {
    repository.deleteById(id);
}

}
```

```

16
17
18 @Entity
19 @Table(name = "produto")
20 public class Produto {
21
22     @Id
23     @GeneratedValue(stra
24     private long id;
25
26     @NotNull
27     @Size(min = 5, max =
28     private String nome;
29
30     @Min(0)
31     private int qtdStoque;
32
33     @ManyToOne
34     @JsonIgnoreProperties("produto")
35     private Categoria categoria;
36
37     public Categoria getCategoria() {
38         return categoria;
39     }
40
41     public void setCategoria(Categoria categoria) {
42         this.categoria = categoria;
43     }
44
45     private BigDecimal valor;
46
47     public long getId() {
48         return id;
49     }
50
51     public void setId(long id) {
52         this.id = id;

```

Estamos informando para o JPA que muitos produtos podem ter apenas um tipo de Categoria, logo eu posso uma tabela com muitos produtos, porém cada um destes produtos poderá pertencer a apenas uma Categoria.

Quando a nossa API devolver este recurso no formato Json, a Anotação `@JsonIgnoreProperties` não permitirá que ao informar a categoria do produto em questão informe também a lista de produtos que a categoria contém, evitando assim uma recursividade infinita

Receberemos um Objeto do tipo Categoria como um atributo categoria dentro da Classe Produto

## Agora testar no Postman para vermos o resultado.

- GetAll em Categorias

```

1 [
2     {
3         "id": 1,
4         "descricao": "Cozinha",
5         "produtos": [
6             {
7                 "id": 2,
8                 "nome": "Fogão",
9                 "qtdStoque": 10,
10                "valor": 100.00
11            }
12        ]
13    },
14    {
15        "id": 2,
16        "descricao": "Sala",
17        "produtos": [
18            {
19                "id": 1,

```

Repare que em cada Categoria temos um Array de Produtos, isto ocorre por que definimos que em Uma Categoria poderíamos ter Muitos Produtos

## • GetAll em Produtos

O mesmo ocorre com os produtos, porém para Cada Produto podemos ter uma Categoria, ou seja, podemos ter Vários produtos com a Categoria Sala.

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
{
  "id": 1,
  "nome": "Radio",
  "qtdStoque": 30,
  "categoria": {
    "id": 2,
    "descricao": "Sala"
  },
  "valor": 50.00
},
{
  "id": 2,
  "nome": "Fogão",
  "qtdStoque": 10,
  "categoria": {
    "id": 1,
    "descricao": "Cozinha"
  }
},

```

## • Post em Produtos

Importante: Agora para inserir um Produto novo, devemos passar qual categoria este Produto pertencia, para fazer isto basta criar um objeto Json Categoria passando apenas o Id

```

1
2
3
4
5
6
7
8
{
  "nome": "Sofá de Canto",
  "qtdStoque": 10,
  "categoria": {
    "id": 2
  },
  "valor": 500.00
}

```

Receberemos o retorno desta forma, mas não se preocupe com a descrição, quando você buscar esta recurso, todos os campos estarão preenchidos.

```

1
2
3
4
5
6
7
8
{
  "id": 4,
  "nome": "Sofá de Canto",
  "qtdStoque": 10,
  "categoria": {
    "id": 2,
    "descricao": null
  }
}

```

- Post em Categorias

The screenshot shows the Postman interface with a POST request to `http://localhost:8080/categorias`. The request body is a JSON object with one item: `"descricao": "Banheiro"`. The response status is `200 OK` with a time of `119ms` and a size of `571 B`. The response body is `{ "id": 3, "descricao": "Banheiro", "produtos": null }`. A callout box points to the `produtos: null` field with the text: **O mesmo serve para Categoria, mesmo contendo uma um atributo produto não precisamos informar.**

**Agora é a sua vez teste os outros EndPoints através do Postman**

**Até a próxima...**